How to Use Methods getResultSet or getUpdateCount to Retrieve the Results?

When you use `execute` method of `Statement` to execute a given SQL statement, it might return multiple result sets and output parameters. In some (uncommon) situations, a single SQL statement may return multiple result sets and/or update counts. Normally you can ignore this unless you are

- executing a stored procedure that you know may return multiple results.
- dynamically executing an unknown SQL string.

The `execute` method excutes an SQL statement and returns a boolean value. When the value is true, the first result returned from the statements is a result set. When the value is false, the first result returned was an update count. You must then use the methods `getResultSet` or `getUpdateCount` to retrieve the result, and `getMoreResults` to move to any subsequent result(s).

When the result of a SQL statement is not a result set, the method `getResultSet` will return `null`. This can mean that the result is an update count or that there are no more results. The only way to find out what the `null` really means in this case is to call the method `getUpdateCount`, which will return an integer. This integer will be the number of rows affected by the calling statement or -1 to indicate either that the result is a result set or that there are no results. If the method getResultSet has already returned null, which means that the result is not a ResultSet object, then a return value of -1 has to mean that there are no more results. In other words, there are no results (or no more results) when the following is true:

```
((stmt.getResultSet() == null) && (stmt.getUpdateCount() == -1))
```

If one has called the method `getResultSet` and processed the `ResultSet` object it returned, it is necessary to call the method `getMoreResults` to see if there is another result set or update count. If getMoreResults returns true, then one needs to again call `getResultSet` to actually retrieve the next result set. As already stated above, if `getResultSet` returns null, one has to call `getUpdateCount` to find out whether null `means` that the result is an update count or that there are no more results.

```
public static void executeStatementSample(Connection con) {
   try {
      String sqlStringWithUnknownResults = "....";
      Statement stmt = con.createStatement();
      boolean results = stmt.execute(sqlStringWithUnknownResults);
      int count = 0;
      do {
         if (results) {
            ResultSet rs = stmt.getResultSet();
            System.out.println("Result set data displayed here.");
         } else {
            count = stmt.getUpdateCount();
            if (count >= 0) {
               System.out.println("DDL or update data displayed here.");
            } else {
               System.out.println("No more results to process.");
            }
         }
         results = stmt.getMoreResults();
      } while (results || count != -1);
      rs.close();
      stmt.close();
   }
   catch (Exception e) {
      e.printStackTrace();
   }
}
```

In the following example, `getMetaData` method of the `Connection` class is used to return a `DatabaseMetadata` object, and then various methods of the `DatabaseMetaData` object are used to display information about the driver, driver version, database name, and database version:

```
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;

public class JDBCDatabaseMetaData {

    private static final String DBURL =
          "jdbc:mysql://localhost:3306/mydb?user=usr&password=sql" +
          "&useUnicode=true&characterEncoding=UTF-8";
    private static final String DBDRIVER = "org.gjt.mm.mysql.Driver";

    static {
        try {
            Class.forName(DBDRIVER).newInstance();
        } catch (Exception e){
            e.printStackTrace();
        }
    }

    private static Connection getConnection()
    {
        Connection connection = null;
        try {
            connection = DriverManager.getConnection(DBURL);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        return connection;
```

```
    }

    public static void main(String[] args) {
      Connection con = getConnection();
      try {
        DatabaseMetaData dbmd = con.getMetaData();
        System.out.println("dbmd:driver version = "
                  + dbmd.getDriverVersion());
        System.out.println("dbmd:driver name = "
                  + dbmd.getDriverName());
        System.out.println("db name = "
                  + dbmd.getDatabaseProductName());
        System.out.println("db ver = "
                  + dbmd.getDatabaseProductVersion());
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }

}
```

To query an unknown result set for information about the columns that it contains, you need to use `ResultSetMetaData` methods to determine the characteristics of the `ResultSets` before you can retrieve data from them. `ResultSetMetaData` methods provide the following types of information:

- `getColumnCount()` method returns the number of columns in the `ResultSet`
- `getTableName()` method returns the qualifier for the underlying table of the `ResultSet`
- `getSchemaName()` method returns the the designated column's table's schema name
- `Information about a column, getColumnName() returns column name, getColumnTypeName() method returns the data type, getColumnDisplaySize() method returns column display length,`

`getPrecision()` method returns the column precision, and `getScale()` method returns scale.

- Whether a column is read-only, nullability, automatically numbered, and so on.

In the following example, it shows various methods of the `ResultSetMetaData` object are used to display information about table and column information within the result set.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Statement;

public class JDBCResultSetMetaData {

    private static final String DBURL =
           "jdbc:mysql://localhost:3306/mydb?user=usr&password=sql"+
           "&useUnicode=true&characterEncoding=UTF-8";
    private static final String DBDRIVER = "org.gjt.mm.mysql.Driver";

    static {
        try {
            Class.forName(DBDRIVER).newInstance();
        } catch (Exception e){
            e.printStackTrace();
        }
    }

    private static Connection getConnection()
    {
        Connection connection = null;
        try {
            connection = DriverManager.getConnection(DBURL);
        }
        catch (Exception e) {
            e.printStackTrace();
```

```
        }
        return connection;
    }

    public static void createEmployees()
    {
        Connection con = getConnection();
        Statement stmt =null;
        String createString;
        createString = "CREATE TABLE  `mydb`.`employees` ("+
           "`EmployeeID` int(10) unsigned NOT NULL default '0',"+
           "`Name` varchar(45) collate utf8_unicode_ci NOT NULL default '',"+
         "`Office` varchar(10) collate utf8_unicode_ci NOT NULL default '',"+
           "`CreateTime` timestamp NOT NULL default CURRENT_TIMESTAMP,"+
           "PRIMARY KEY  (`EmployeeID`)"+
           ") ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;";
        try {
            stmt = con.createStatement();
            stmt.executeUpdate(createString);
        } catch(SQLException ex) {
            System.err.println("SQLException: " + ex.getMessage());
        }
        finally {
            if (stmt != null) {
                try {
                    stmt.close();
                } catch (SQLException e) {
                    System.err.println("SQLException: " + e.getMessage());
                }
            }
            if (con != null) {
                try {
                    con.close();
                } catch (SQLException e) {
                    System.err.println("SQLException: " + e.getMessage());
                }
            }
        }
    }
    private static void dropEmployees()
    {
        Connection con = getConnection();
```

```
        Statement stmt =null;
        String createString;
        createString = "DROP TABLE IF EXISTS `mydb`.`employees`;";
        try {
            stmt = con.createStatement();
            stmt.executeUpdate(createString);
        } catch(SQLException ex) {
            System.err.println("SQLException: " + ex.getMessage());
        }
        finally {
            if (stmt != null) {
                try {
                    stmt.close();
                } catch (SQLException e) {
                    System.err.println("SQLException: " + e.getMessage());
                }
            }
            if (con != null) {
                try {
                    con.close();
                } catch (SQLException e) {
                    System.err.println("SQLException: " + e.getMessage());
                }
            }
        }
    }

    private static void insertEmployee()
    {
        Connection con = getConnection();
        Statement stmt = null;

        try {
            stmt = con.createStatement();

            stmt.addBatch("INSERT INTO employees(EmployeeID, Name, Office) "
                    + "VALUES(1001, 'David Walker', 'HQ101')");

            stmt.addBatch("INSERT INTO employees(EmployeeID, Name, Office) "
                    + "VALUES(1002, 'Paul Walker', 'HQ202')");

            stmt.addBatch("INSERT INTO employees(EmployeeID, Name, Office) "
```

```
                    +  "VALUES(1003, 'Scott Warner', 'HQ201')");

            int [] updateCounts = stmt.executeBatch();

        } catch (SQLException e) {
            System.err.println("SQLException: " + e.getMessage());
        }
        finally {
            if (stmt != null) {
                try {
                    stmt.close();
                } catch (SQLException e) {
                    System.err.println("SQLException: " + e.getMessage());
                }
            }
            if (con != null) {
                try {
                    con.close();
                } catch (SQLException e) {
                    System.err.println("SQLException: " + e.getMessage());
                }
            }
        }
    }
    public static void showEmployeeInfo()
    {
        Connection con = getConnection();
        Statement stmt = null;

        try {
            stmt = con.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT * from Employees");
            ResultSetMetaData rsmd = rs.getMetaData();
            int cols = rsmd.getColumnCount();
            System.out.println("Column Count is " + cols);
            for (int i = 1; i <= cols; i++) {
               System.out.println("\nNAME: " + rsmd.getColumnName(i) + "\n" +
                     "TYPE: " + rsmd.getColumnTypeName(i) + "\n" +
                      "TABLE: " + rsmd.getTableName(i)+"\n" +
                      "Schema: " + rsmd.getSchemaName(i) + "\n" +
                      "Scale: " + rsmd.getScale(i) + "\n" +
                       "Length: " + rsmd.getColumnDisplaySize(i) + "\n" +
```

```
                        "Precision: " + rsmd.getPrecision(i));
            }
            rs.close();
        } catch (SQLException e) {
            System.err.println("SQLException: " + e.getMessage());
        }
        finally {
            if (stmt != null) {
                try {
                    stmt.close();
                } catch (SQLException e) {
                    System.err.println("SQLException: " + e.getMessage());
                }
            }
            if (con != null) {
                try {
                    con.close();
                } catch (SQLException e) {
                    System.err.println("SQLException: " + e.getMessage());
                }
            }
        }

    }
    public static void main(String[] args) {
        dropEmployees();
        createEmployees();
        insertEmployee();
        showEmployeeInfo();
    }

}
```

In the following example, it shows various methods of the `ParameterMetaData` object are used to display information about the type and mode of the parameters that are contained within a stored procedure.

```
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DatabaseMetaData;
```

```
import java.sql.DriverManager;
import java.sql.ParameterMetaData;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;

public class JDBCPrepareMetaData {

    private static final String DBURL =
            "jdbc:mysql://localhost:3306/mydb?user=usr&password=sql"
             +"&useUnicode=true&characterEncoding=UTF-8";
    private static final String DBDRIVER = "org.gjt.mm.mysql.Driver";

    static {
        try {
            Class.forName(DBDRIVER).newInstance();
        } catch (Exception e){
            e.printStackTrace();
        }
    }

    private static Connection getConnection()
    {
        Connection connection = null;
        try {
            connection = DriverManager.getConnection(DBURL);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        return connection;
    }

    public static void createEmployees()
    {
        Connection con = getConnection();
        Statement stmt =null;
        String createString;
        createString = "CREATE TABLE  `mydb`.`employees` ("+
```

```
          "`EmployeeID` int(10) unsigned NOT NULL default '0',"+
          "`Name` varchar(45) collate utf8_unicode_ci NOT NULL default '',"+
        "`Office` varchar(10) collate utf8_unicode_ci NOT NULL default '',"+
          "`CreateTime` timestamp NOT NULL default CURRENT_TIMESTAMP,"+
          "PRIMARY KEY  (`EmployeeID`)"+
          ") ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;";
        try {
            stmt = con.createStatement();
            stmt.executeUpdate(createString);
        } catch(SQLException ex) {
            System.err.println("SQLException: " + ex.getMessage());
        }
        finally {
            if (stmt != null) {
                try {
                    stmt.close();
                } catch (SQLException e) {
                    System.err.println("SQLException: " + e.getMessage());
                }
            }
            if (con != null) {
                try {
                    con.close();
                } catch (SQLException e) {
                    System.err.println("SQLException: " + e.getMessage());
                }
            }
        }
    }
    private static void dropEmployees()
    {
        Connection con = getConnection();
        Statement stmt =null;
        String createString;
        createString = "DROP TABLE IF EXISTS `mydb`.`employees`;";
        try {
            stmt = con.createStatement();
            stmt.executeUpdate(createString);
        } catch(SQLException ex) {
            System.err.println("SQLException: " + ex.getMessage());
        }
        finally {
```

```
            if (stmt != null) {
                try {
                    stmt.close();
                } catch (SQLException e) {
                    System.err.println("SQLException: " + e.getMessage());
                }
            }
            if (con != null) {
                try {
                    con.close();
                } catch (SQLException e) {
                    System.err.println("SQLException: " + e.getMessage());
                }
            }
        }
    }

    private static void insertEmployee()
    {
        Connection con = getConnection();
        Statement stmt = null;

        try {
            stmt = con.createStatement();

            stmt.addBatch("INSERT INTO employees(EmployeeID, Name, Office) "
                   + "VALUES(1001, 'David Walker', 'HQ101')");

            stmt.addBatch("INSERT INTO employees(EmployeeID, Name, Office) "
                    + "VALUES(1002, 'Paul Walker', 'HQ202')");

            stmt.addBatch("INSERT INTO employees(EmployeeID, Name, Office) "
                   + "VALUES(1003, 'Scott Warner', 'HQ201')");

            int [] updateCounts = stmt.executeBatch();

        } catch (SQLException e) {
            System.err.println("SQLException: " + e.getMessage());
        }
        finally {
            if (stmt != null) {
                try {
```

```
                stmt.close();
            } catch (SQLException e) {
                System.err.println("SQLException: " + e.getMessage());
            }
        }
        if (con != null) {
            try {
                con.close();
            } catch (SQLException e) {
                System.err.println("SQLException: " + e.getMessage());
            }
        }
    }
}
private    static void dropProcedure() {
    Connection con = getConnection();
    Statement stmt =null;
    try {
        stmt = con.createStatement();
        stmt.execute("DROP PROCEDURE IF EXISTS `mydb`.`WhoAreThey`");
    } catch(SQLException ex) {
        System.err.println("SQLException: " + ex.getMessage());
    }
    finally {
        if (stmt != null) {
            try {
                stmt.close();
            } catch (SQLException e) {
                System.err.println("SQLException: " + e.getMessage());
            }
        }
        if (con != null) {
            try {
                con.close();
            } catch (SQLException e) {
                System.err.println("SQLException: " + e.getMessage());
            }
        }
    }

}
```

```
private    static void createProcedure() {
    Connection con = getConnection();
    Statement stmt =null;
    try {
        stmt = con.createStatement();
           stmt.execute("CREATE PROCEDURE `WhoAreThey`(" +
                   "OUT error VARCHAR(128)," +
                   "IN office VARCHAR(10)) " +
                   "BEGIN "+
                   "SET error = NULL; "+
                   "IF office IS NULL THEN "+
                   "SET error = 'You need to pass in an office number'; "+
                   "ELSE "+
                   "  SELECT EmployeeID, Name FROM employees " +
                   " WHERE office = office; "+
                   "END IF; "+
                   "END");

    } catch(SQLException ex) {
        System.err.println("SQLException: " + ex.getMessage());
    }
    finally {
        if (stmt != null) {
            try {
                stmt.close();
            } catch (SQLException e) {
                System.err.println("SQLException: " + e.getMessage());
            }
        }
        if (con != null) {
            try {
                con.close();
            } catch (SQLException e) {
                System.err.println("SQLException: " + e.getMessage());
            }
        }
    }

}

public static void getParameterMetaData() {
    Connection con = getConnection();
```

```
PreparedStatement stmt = null;

try {
    stmt = con.prepareStatement("INSERT INTO " +
          employees(EmployeeID, Name, Office) " +
            "VALUES(?,?,?)");

    ParameterMetaData pmd = stmt.getParameterMetaData();
    int count = pmd.getParameterCount();
    for (int i = 1; i <= count; i++) {
        System.out.println("Name: " + pmd.getParameterType(i));
        System.out.println("TYPE: " + pmd.getParameterTypeName(i));
        String mode = "";
        switch(pmd.getParameterMode(i)) {
             case ParameterMetaData.parameterModeIn:
                 mode = "IN";
                 break;
             case ParameterMetaData.parameterModeInOut:
                 mode = "IN/OUT";
                 break;
             case ParameterMetaData.parameterModeOut:
                 mode = "OUT";
                 break;
             case ParameterMetaData.parameterModeUnknown:
                 mode = "UNKNOWN";
                 break;

        }
        System.out.println("MODE: " + mode);
    }

} catch (SQLException e) {
    System.err.println("SQLException: " + e.getMessage());
}
finally {
    if (stmt != null) {
        try {
            stmt.close();
        } catch (SQLException e) {
            System.err.println("SQLException: " + e.getMessage());
        }
    }
```

```
            if (con != null) {
                try {
                    con.close();
                } catch (SQLException e) {
                    System.err.println("SQLException: " + e.getMessage());
                }
            }
        }

    }

    public static void getProcedureParameterMetaData() {
        Connection con = getConnection();
        CallableStatement cs = null;
        try {
            cs = con.prepareCall("{call WhoAreThey(?,?)}");
            ParameterMetaData pmd = cs.getParameterMetaData();
            int count = pmd.getParameterCount();
            for (int i = 1; i <= count; i++) {
                System.out.println("Name: " + pmd.getParameterType(i));
                System.out.println("TYPE: " + pmd.getParameterTypeName(i));
                String mode = "";
                switch(pmd.getParameterMode(i)) {
                     case ParameterMetaData.parameterModeIn:
                         mode = "IN";
                         break;
                     case ParameterMetaData.parameterModeInOut:
                         mode = "IN/OUT";
                         break;
                     case ParameterMetaData.parameterModeOut:
                         mode = "OUT";
                         break;
                     case ParameterMetaData.parameterModeUnknown:
                         mode = "UNKNOWN";
                         break;

                }
                System.out.println("MODE: " + mode);
            }

        } catch (SQLException e) {
            System.err.println("SQLException: " + e.getMessage());
```

```
        }
        finally {
            if (cs != null) {
                try {
                    cs.close();
                } catch (SQLException e) {
                    System.err.println("SQLException: " + e.getMessage());
                }
            }
            if (con != null) {
                try {
                    con.close();
                } catch (SQLException e) {
                    System.err.println("SQLException: " + e.getMessage());
                }
            }
        }

    }

    public static void main(String[] args) {
        dropProcedure();
        dropEmployees();
        createEmployees();
        insertEmployee();
        getParameterMetaData();
        System.out.println("Call Procedure.....");
        createProcedure();
        getProcedureParameterMetaData();
    }

}
```

A default ResultSet object is not updatable and has a cursor that moves forward only. Thus, you can iterate through it only once and only from the first row to the last row. It is possible to produce ResultSet objects that are scrollable and/or updatable. An updatable result set allows modification to data in a table through the result set. The following code makes a result set that is scrollable and insensitive to updates by others:

```
try {
    // Create a statement that will return updatable result sets
    Statement stmt = connection.createStatement(
                ResultSet.TYPE_SCROLL_SENSITIVE,
                ResultSet.CONCUR_UPDATABLE);

    //Primary key EmployeeID must be specified
    //so that the result set is updatable
    ResultSet resultSet = stmt.executeQuery(
                "SELECT EmployeeID, Name, Office FROM employees");
} catch (SQLException e) {
}
```

The updatable result set may be used few ways:

- to update a column value in the current row. In a scrollable `ResultSet` object, the cursor can be moved backwards and forwards, to an absolute position, or to a position relative to the current row. The following code fragment updates the Office column in the fifth row of the ResultSet object rs and then uses the method updateRow to update the data source table from which rs was derived.

```
rs.absolute(5); // moves the cursor to the fifth row of rs
rs.updateString("Office", "HQ222"); // updates the
// Office column of row 5 to be HQ222
rs.updateRow(); // updates the row in the data source
```

- to insert column values into the insert row. An updatable ResultSet object has a special row associated with it that serves as a staging area for building a row to be inserted. The following code fragment moves the cursor to the insert row, builds a three-column row, and inserts it into rs and into the data source table using the method insertRow.

```
rs.moveToInsertRow(); // moves cursor to the insert row
rs.updateInt("EmployeeID", 1001);
rs.updateString("Name", "Divad Walker");
rs.updateString("Office", "HQ101");
```

```
rs.insertRow();
rs.moveToCurrentRow();
```

- to delete a row. The following code fragment moves to the first row of the ResultSet object rs and then uses the method deleteRow to delete the data source table from which rs was derived.

```
rs.first(); // moves cursor to the deleting row
rs.deleteRow();
```

A `ResultSet` object is automatically closed when the `Statement` object that generated it is closed, re-executed, or used to retrieve the next result from a sequence of multiple results.

`cancelRowUpdates` cancels the updates made to the current row in this `ResultSet` object. This method may be called after calling an updater method(s) and before calling the method `updateRow` to roll back the updates made to a row. If no updates have been made or `updateRow` has already been called, this method has no effect.

The following is an example for this topic:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class JDBCUpdatableRS {

    private static final String DBURL =
               "jdbc:mysql://localhost:3306/mydb?user=usr&password=sql" +
              "&useUnicode=true&characterEncoding=UTF-8";
    private static final String DBDRIVER = "org.gjt.mm.mysql.Driver";

    static {
        try {
```

```
            Class.forName(DBDRIVER).newInstance();
        } catch (Exception e){
            e.printStackTrace();
        }
    }

    private static Connection getConnection()
    {
        Connection connection = null;
        try {
            connection = DriverManager.getConnection(DBURL);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        return connection;
    }

    public static void createEmployees()
    {
        Connection con = getConnection();
        Statement stmt =null;
        String createString;
        createString = "CREATE TABLE  `mydb`.`employees` ("+
         "`EmployeeID` int(10) unsigned NOT NULL default '0',"+
         "`Name` varchar(45) collate utf8_unicode_ci NOT NULL default '',"+
        "`Office` varchar(10) collate utf8_unicode_ci NOT NULL default '',"+
        "`CreateTime` timestamp NOT NULL default CURRENT_TIMESTAMP,"+
        "PRIMARY KEY  (`EmployeeID`)"+
       ") ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;";
        try {
            stmt = con.createStatement();
               stmt.executeUpdate(createString);
        } catch(SQLException ex) {
            System.err.println("SQLException: " + ex.getMessage());
        }
        finally {
            if (stmt != null) {
                try {
                    stmt.close();
                } catch (SQLException e) {
                    System.err.println("SQLException: " + e.getMessage());
```

```
                }
            }
            if (con != null) {
                try {
                    con.close();
                } catch (SQLException e) {
                    System.err.println("SQLException: " + e.getMessage());
                }
            }
        }
    }
    private static void dropEmployees()
    {
        Connection con = getConnection();
        Statement stmt =null;
        String createString;
        createString = "DROP TABLE IF EXISTS `mydb`.`employees`;";
        try {
            stmt = con.createStatement();
            stmt.executeUpdate(createString);
        } catch(SQLException ex) {
            System.err.println("SQLException: " + ex.getMessage());
        }
        finally {
            if (stmt != null) {
                try {
                    stmt.close();
                } catch (SQLException e) {
                    System.err.println("SQLException: " + e.getMessage());
                }
            }
            if (con != null) {
                try {
                    con.close();
                } catch (SQLException e) {
                    System.err.println("SQLException: " + e.getMessage());
                }
            }
        }
    }

    public static void showEmployee() {
```

```
        Connection con = getConnection();
        Statement stmt =null;
        try {
            stmt = con.createStatement();
               ResultSet rs = stmt.executeQuery("Select * from employees "
                       + where EmployeeID=1001");
               if (rs.next()) {
                   System.out.println("EmployeeID : " +
                            rs.getInt("EmployeeID"));
                   System.out.println("Name : " + rs.getString("Name"));
                   System.out.println("Office : " + rs.getString("Office"));
               }
               else {
                   System.out.println("No Specified Record.");
               }
               rs.close();
        } catch(SQLException ex) {
            System.err.println("SQLException: " + ex.getMessage());
        }
        finally {
            if (stmt != null) {
                try {
                    stmt.close();
                } catch (SQLException e) {
                    System.err.println("SQLException: " + e.getMessage());
                }
            }
            if (con != null) {
                try {
                    con.close();
                } catch (SQLException e) {
                    System.err.println("SQLException: " + e.getMessage());
                }
            }
        }

    }
    public static void insertEmployee() {
        Connection con = getConnection();
        Statement stmt =null;
        String sqlString = "SELECT EmployeeID, Name, " +
             " Office FROM employees;";
```

```
        try {
            stmt = con.createStatement(
                    ResultSet.TYPE_SCROLL_SENSITIVE,
                    ResultSet.CONCUR_UPDATABLE);
            ResultSet rs = stmt.executeQuery(sqlString);

            //Check the result set is an updatable result set
            int concurrency = rs.getConcurrency();
            if (concurrency == ResultSet.CONCUR_UPDATABLE) {
                rs.moveToInsertRow();
                rs.updateInt(1, 1001);
                rs.updateString(2, "Divad Walker");
                rs.updateString(3, "HQ101");
                rs.insertRow();
                rs.moveToCurrentRow();
            } else {
            System.out.println("ResultSet is not an updatable result set.");
            }
            rs.close();
        } catch(SQLException ex) {
            System.err.println("SQLException: " + ex.getMessage());
        }
        finally {
            if (stmt != null) {
                try {
                    stmt.close();
                } catch (SQLException e) {
                    System.err.println("SQLException: " + e.getMessage());
                }
            }
            if (con != null) {
                try {
                    con.close();
                } catch (SQLException e) {
                    System.err.println("SQLException: " + e.getMessage());
                }
            }
        }
    }

    public static void updateEmployee(){
        Connection con = getConnection();
```

```
        Statement stmt =null;"
        String sqlString = "SELECT EmployeeID, Name, Office " +
            " FROM employees WHERE EmployeeID=1001";
        try {
            stmt = con.createStatement(
                    ResultSet.TYPE_SCROLL_SENSITIVE,
                    ResultSet.CONCUR_UPDATABLE);
            ResultSet rs = stmt.executeQuery(sqlString);

            //Check the result set is an updatable result set
            int concurrency = rs.getConcurrency();
            if (concurrency == ResultSet.CONCUR_UPDATABLE) {
                rs.first();
                rs.updateString("Office", "HQ222");
                rs.updateRow();
            } else {
            System.out.println("ResultSet is not an updatable result set.");
            }
            rs.close();
        } catch(SQLException ex) {
            System.err.println("SQLException: " + ex.getMessage());
        }
        finally {
            if (stmt != null) {
                try {
                    stmt.close();
                } catch (SQLException e) {
                    System.err.println("SQLException: " + e.getMessage());
                }
            }
            if (con != null) {
                try {
                    con.close();
                } catch (SQLException e) {
                    System.err.println("SQLException: " + e.getMessage());
                }
            }
        }
    }

    public static void deleteEmployee(){
        Connection con = getConnection();
```

```
    Statement stmt =null;
    String sqlString = "SELECT EmployeeID, Name, Office " +
          " FROM employees WHERE EmployeeID=1001";
    try {
        stmt = con.createStatement(
                ResultSet.TYPE_SCROLL_SENSITIVE,
                ResultSet.CONCUR_UPDATABLE);
        ResultSet rs = stmt.executeQuery(sqlString);

        //Check the result set is an updatable result set
        int concurrency = rs.getConcurrency();
        if (concurrency == ResultSet.CONCUR_UPDATABLE) {
            rs.first();
            rs.deleteRow();
        } else {
        System.out.println("ResultSet is not an updatable result set.");
        }
        rs.close();
    } catch(SQLException ex) {
        System.err.println("SQLException: " + ex.getMessage());
    }
    finally {
        if (stmt != null) {
            try {
                stmt.close();
            } catch (SQLException e) {
                System.err.println("SQLException: " + e.getMessage());
            }
        }
        if (con != null) {
            try {
                con.close();
            } catch (SQLException e) {
                System.err.println("SQLException: " + e.getMessage());
            }
        }
    }
}

public static void main(String[] args) {
    dropEmployees();
    createEmployees();
```

```
        insertEmployee();
        System.out.println("\nAfter inserting a Record ...");
        showEmployee();
        updateEmployee();
        System.out.println("\nAfter updating a Record ...");
        showEmployee();
        deleteEmployee();
        System.out.println("\nAfter deleting a Record ...");
        showEmployee();
    }

}
```

This code example shows how to convert HashSet to ArrayList:

```
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class HashSetToArrayList {

  public static void main(String[] args) {

    Set<String> hashset = new HashSet<String>();
    hashset.add("A");
    hashset.add("B");
    hashset.add("C");
    List<String> list = new ArrayList<String>(hashset);
    System.out.println(list.toString());

  }
}
```

This code example shows how to convert ArrayList to HashSet:

```
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
```

```
import java.util.Set;

public class ArrayListToHashSet {

  public static void main(String[] args) {
    List<String> list = new ArrayList<String>();
    list.add(null);
    list.add("A");
    list.add("B");
    Set<String> hashset = new HashSet<String>(list);
    list = new ArrayList<String>(hashset);
    System.out.println(list.toString());
  }
}
```

The above code example also can be used to remove duplicate items from an ArrayList.

```
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class ArrayListToHashSet {

        public static void main(String[] args) {
                List list = new ArrayList();
                list.add(null);
                list.add("A");
                list.add("A");
                list.add("A");
                list.add("A");
                list.add("B");
                list.add("B");
                list.add("B");
                list.add("C");
                Set hashset = new HashSet(list);
                list = new ArrayList(hashset);
                System.out.println(list.toString());
        }
```

```
}
```

This code example shows how to list all system properties, get/set the value of a system property:

```
import java.util.Enumeration;
import java.util.Properties;

public class ListingAllSystemProperties {

  public static void main(String[] args) {

    //List All System Properties
    Properties props = System.getProperties();
    Enumeration enumeration = props.keys();
    while (enumeration.hasMoreElements()) {
      String propName = (String) enumeration.nextElement();
      String propValue = (String)props.get(propName);
      System.out.println(propName + " = " + propValue);
    }


    // Set a system property
    String previousValue = System.setProperty("myjava.version", "5.0");
    //Get a system property
    String version = System.getProperty("myjava.version");
    System.out.println("myjava.version=" + version);

  }
}
```

Here is an example to get the working directory (which is the location in the file system from where the java command was invoked):

```
    String curDir = System.getProperty("user.dir");
```

A system property can be set or overridden by specifying the -D option to the java command when running your program.

```
java -Dmyjava.version="5.0" MyApplication
```

public interface **FilenameFilter** is an interface that declares single method. Instances of classes that implement this interface are used to filter filenames. These instances are used to filter directory listings in the `list` method of class `File`, and by the Abstract Window Toolkit's file dialog component.

There is one and only one method in the interface, `public boolean accept(File directory, String filename)`. The method returns `true` if and only if the filename should be included in the file list; `false` otherwise.

The *FilenameFilter* is an interface and you must implement this interface in your class. Here is a sample implemeting the method which returns all java files in given directory, the file filter only accepts files ending with ".java".

```
public static String[] getFileNames(String dirName) throws IOException{

  File dir = new File(dirName);

  FilenameFilter filter = new FilenameFilter() {
        public boolean accept(File dir, String name) {
          return name.endsWith(".java"));
        }
      };

  return dir.list(filter);
}
```

Here an example uses File class to retrive all files and subdirectories under the root.

```
  public static void main(String[] argv){
    File dir = new File("c:\\");
    String[] children = dir.list();
    if (children != null) {
      for (String filename: children) {
        out.println(filename);
      }
    }
    else {
      out.println("No File Found.");
    }
}
```

Here is an example showing how to return subdirectories only. From *java.io.File* Java API doc, a list of files can also be retrieved as array of File objects.

```
class DirectoryFileFilter implements FileFilter
{
  public boolean accept(File file) {
        return file.isDirectory();
  }
}
public class ListSubdirectory {
  public static void main(String[] argv){

    File dir = new File("c:\\");
    File[] files = dir.listFiles(new DirectoryFileFilter());
    if (files != null) {
      for (File file: files) {
        out.println(file.getName());
      }
    }
    else {
      out.println("No Subdirectory Found.");
    }
  }
}
```

Here is an example to retrive a set of document file which given file extensions.

```
class MyDocFileFilter implements FileFilter
{
  private final String[] myDocumentExtensions
              = new String[] {".txt", ".doc", ".html", "htm"};

  public boolean accept(File file) {

        if (!file.isFile())
                return false;

    for (String extension : myDocumentExtensions) {
      if (file.getName().toLowerCase().endsWith(extension))
        return true;
    }
    return false;
  }
}

public class ListMyDocuments {

  public static void main(String [] argv){
    File dir = new File("c:\\");
    File[] files = dir.listFiles(new MyDocFileFilter());
    if (files != null) {
      for (File file: files) {
        out.println(file.getName());
      }
    }
    else {
      out.println("No File Found.");
    }
  }

}
```

More than often we need to manipulate a string, substitute characters inside

a string. In Java, the String class, there are a couple of methods that we can use to complete this task.

1. public String replace(char oldChar, char newChar)

This method "returns a new string resulting from replacing all occurrences of oldChar in this string with newChar."
Both oldChar and newChar are single char.
For example, String a = "This is a cat.";
String b = a.replace('c', 'r');
b has the value of "This is a rat.".

2. public String replace(CharSequence target, CharSequence replacement)

This method "replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence."
String c = "This is a cat.";
String d = c.replace("ca", "rabbi");
d has the value of "This is a rabbit.".

Note: this method only works with replacing a sub string with a string, it doesn't work with regex. For example:

String e = "This is a cat.";
String f = e.replace("\\s", "b");
f is still "This is a cat.". Because the function is looking for a string like "\s" to replace.

Sometimes, you need to replace many characters, regex comes handy. For example, you want to create a clean url for an article by using its title, but a

title can contain special characters that may break your url. One way to do this is to strip out all the non-alphanumerics from the string. Then you will need to use the following method to handle regex target.

3. public String replaceAll(String regex, String replacement)

This method "replaces each substring of this string that matches the given regular expression with the given replacement."

```
String title = "Since When Do Politicians \"Care\" About Newspapers?";
String result = title.replaceAll("[^a-zA-Z0-9\\s]", "");
result = result.replaceAll("\\s", "-"); // replace white spaces to "-".
The result is "Since-When-Do-Politicians-Care-About-Newspapers";
```

4. public String replaceFirst(String regex, String replacement)

This method "replaces the first substring of this string that matches the given regular expression with the given replacement."

```
String title = "Since When Do Politicians Care About Newspapers?";
String result = title.replaceAll("[^a-zA-Z0-9\\s]", "");
result = result.replaceAll("\\s", "-"); // replace white spaces to "-".
The result is "Since-WhenDo Politicians Care About Newspapers?".
```

An *enum* type, also called enumeration type, is a type whose fields consist of a fixed set of constants. The purpose of using enum type is to enforce type safety.

While java.lang.Enum is an abstract class, it is the common base class of all Java language enumeration types. The definition of Enum is:

```
public abstract class Enum>
extends Object
implements Comparable, Serializable
```

All *enum* types implicitly extend java.lang.Enum.

The *enum* is a special reference type, it is not a class by itself, but more like a category of classes that extends from the same base class Enum. Any type declared by the key word "enum" is a different class. They easiest way to declare a enum type is like:

```
public enum Season {
   SPRING, SUMMER, AUTUM, WINTER
}
```

or

```
public enum Day {
   SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
   THURSDAY, FRIDAY, SATURDAY
}
```

Here Season and Day are both enum, but they are different type. All the constants in a enum type is of the same type. For example, SPRING and SUMMER are both of type Season.

Since all enum types All enum types implicitly extend java.lang.Enum, all the methods defined in the Enum class are available to all enum types, such as the

String name() method, which returns the name of this enum constant

int ordinal() method, which returns the ordinal of this enumeration constant, starts from 0

For more information, refer to SCJP Study Guide: Enums

Note: Since Java does not support multiple inheritance, an enum cannot extend anything else.

Sometime, you need to lookup an enum from its value (may be a integer, string or other types). This reverse lookup can be easily implemented by using a static java.util.Map inside your enum class. For example,

```
import java.util.HashMap;
import java.util.Map;

public enum Day {
     SUNDAY(0),
     MONDAY(1),
     TUESDAY(2),
     WEDNESDAY(3),
     THURSDAY(4),
     FRIDAY(5),
     SATURDAY(6);

     private static final Map lookup =

                        new HashMap();

     static {
         //Create reverse lookup hash map
         for(Day d : Day.values())
             lookup.put(d.getDayValue(), d);
     }

     private int dayValue;
```

```
     private Day(int dayValue) {
          this.dayValue = dayValue;
     }

     public int getDayValue() { return dayValue; }

     public static Day get(int dayValue) {

        //the reverse lookup by simply getting
        //the value from the lookup HsahMap.

          return lookup.get(dayValue);
     }
}
```

For example,

```
class ClassUtil {

    public String fullyQualifiedName;

    public ClassUtil(String fullyQualifiedName) {
        super();
        if (fullyQualifiedName == null)
            this.fullyQualifiedName = "";
        else {
            this.fullyQualifiedName = fullyQualifiedName.trim();
        }
    }

    public ClassUtil(Class c) {

        super();
        this.fullyQualifiedName = c.getName();

    }

    public String getFullClassName() {
```

```
        return this.fullyQualifiedName;
    }

    public String getPackageName() {
        int lastDot = fullyQualifiedName.lastIndexOf ('.');
        return (lastDot<=0)?"":fullyQualifiedName.substring (0, lastDot);
    }

    public String getClassName() {
        int lastDot = fullyQualifiedName.lastIndexOf ('.');
        return (lastDot<=0)?fullyQualifiedName:fullyQualifiedName.substring (+
    }


}

public class StackTraceDisplay {

    public static void main(String[] args) {
        doFun1();
    }

    public static void doFun1(){
        doFun2();
    }

    public static void doFun2(){
        displayStackTrace(Thread.currentThread().getStackTrace());
    }

    public static void displayStackTrace(StackTraceElement e[]) {

        for (StackTraceElement s : e) {
            if (s.getMethodName().equals("getStackTrace"))
                continue;
            System.out.println ("Filename: " + s.getFileName());
            System.out.println ("Line number: " + s.getLineNumber());
            ClassUtil cu = new ClassUtil(s.getClassName());
            System.out.println ("Package name: " + cu.getPackageName());
            System.out.println ("Full Class name: " + cu.getFullClassName());
            System.out.println ("Classlass name: " + cu.getClassName());
```

```
            System.out.println ("Method name: " + s.getMethodName());
            System.out.println ("Native method?: " + s.isNativeMethod());
            System.out.println ("toString(): " + s.toString());
            System.out.println ("");
        }

    }

}
```

The class java.lang.ProcessBuilder, in Java 1.5, is used to create operating system processes. Each process builder manages these process attributes : (See Java API Document)

- a command, a list of strings which signifies the external program file to be invoked and its arguments, if any. Which string lists represent a valid operating system command is system-dependent. For example, it is common for each conceptual argument to be an element in this list, but there are operating systems where programs are expected to tokenize command line strings themselves - on such a system a Java implementation might require commands to contain exactly two elements.
- an environment, which is a system-dependent mapping from variables to values. The initial value is a copy of the environment of the current process (see System.getenv()).
- a working directory. The default value is the current working directory of the current process, usually the directory named by the system property user.dir.
- a redirectErrorStream property. Initially, this property is false, meaning that the standard output and error output of a subprocess are sent to two separate streams, which can be accessed using the Process.getInputStream() and Process.getErrorStream() methods. If the

value is set to true, the standard error is merged with the standard output. This makes it easier to correlate error messages with the corresponding output. In this case, the merged data can be read from the stream returned by Process.getInputStream(), while reading from the stream returned by Process.getErrorStream() will get an immediate end of file.

The java.lang.ProcessBuilder and java.lang.Process classes are available for executing and communicating with external programs. With an instance of the java.lang.ProcessBuilder class, it can execute an external program and return an instance of a subclass of java.lang.Process. The class Process provides methods for performing input from the process, performing output to the process, waiting for the process to complete, checking the exit status of the process, and destroying (killing) the process.

```
public class ProcessBuildDemo {

    public static void main(String [] args) throws IOException {

        String[] command = {"CMD", "/C", "dir"};
        ProcessBuilder probuilder = new ProcessBuilder( command );


        //You can set up your work directory
        probuilder.directory(new File("c:\\xyzwsdemo"));

        Process process = probuilder.start();

        //Read out dir output
        InputStream is = process.getInputStream();
        InputStreamReader isr = new InputStreamReader(is);
        BufferedReader br = new BufferedReader(isr);
        String line;
        System.out.printf("Output of running %s is:\n",
                Arrays.toString(command));
```

```
        while ((line = br.readLine()) != null) {
            System.out.println(line);
        }

        //Wait to get exit value
        try {
            int exitValue = process.waitFor();
            System.out.println("\n\nExit Value is " + exitValue);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

Did you ever encounter an runtime exception saying a method not found? How can the code compile if the method is not defined? Or after you made some changes to the code, you don't see it takes effect, even after you clean built, and redeploy for a couple of times? When these situations occurs, besides trying to pull your hair out, you can in fact find out which jar the class in trouble is loaded from. More than often, there is another jar file sitting the the classpath which contains the exact same class.

If you are handy in shell scripts, you may come up with a simple script using "find" to track down all the jar files in the class path that contains the class you are looking for.

If you are not a script guru, you can try the following:

Put this code snippet inside any one of the method of any one of the class:

Class klass = ClassInTrouble.class;

CodeSource codeSource = klass.getProtectionDomain().getCodeSource();

if (codeSource != null) {

```
    System.out.println(codeSource.getLocation());

  }
```

When the method that contains this sniplet is called, it will print out something like:

file://c:/dev/source/lib/myjar.jar

Will the codeSource be null? The answer is yes. When a jar file is loaded by the system class loader, it's codeSource will be null. What jars will be loaded by the system class loader? the rule of thumb is that all the jars in the class path (not the ones you package in your application) will be loaded by the system class loader. Obviously the above code can't used to find out which jar a class is loaded from, if the jar is loaded by the system class loader. You can use the "verbose" java command line argument when you start the application:

java -verbose app

it will print out every class in each jar the system class loader loads, ie:

```
[Opened C:\dev\bea\JDK160~1\jre\lib\rt.jar]
[Loaded java.lang.Object from C:\dev\bea\JDK160~1\jre\lib\rt.jar]
[Loaded java.io.Serializable from C:\dev\bea\JDK160~1\jre\lib\rt.jar]
[Loaded java.lang.Comparable from C:\dev\bea\JDK160~1\jre\lib\rt.jar]
[Loaded java.lang.CharSequence from C:\dev\bea\JDK160~1\jre\lib\rt.jar]
[Loaded java.lang.String from C:\dev\bea\JDK160~1\jre\lib\rt.jar]
[Loaded java.lang.reflect.GenericDeclaration from
C:\dev\bea\JDK160~1\jre\lib\rt.jar]
......
```

How to accessing a resource within a jar file?

You access some resources (images, xml file or properties file) that are inside a jar. For example, you can retrieve an image with the following way:

```
InputStream in = this.getClass().getClassLoader()
                .getResourceAsStream("com/linar/java2com/plus.gif");
int c;
ByteArrayOutputStream byteArrayOutputStream = new
ByteArrayOutputStream();
while ((c = in.read()) != -1) {
   byteArrayOutputStream.write((char) c);
}
```

Here is another example using the getResource() method that takes an URL parameter to retrieve an image:

```
URL url = this.getClass().getClassLoader()
          .getResource("com/linar/java2com/plus.gif");
ImageIcon image = (new ImageIcon(url));
```

The Class.forName() method allows you to map a case-sensitive class name to the Class instance representing gaven class. Then you can invoke its newInstance() to create an instance of that class. For Example:

```
Class tc = Class.forName("com.xyzws.common.Type");
Tyep myType = tc.newInstance();
```

In the event that the Class could not be found, resolved, verified, or loaded, Class.forName
throws one of several different Exceptions, all of which are listed in the javadoc page for
java.lang.Class.

Another example to check where class's jar file is:

```
try {
   String qualifiedClassName="org.xbill.DNS.DSRecord";
   Class qc = Class.forName(qualifiedClassName);
   CodeSource source = qc.getProtectionDomain().getCodeSource();
   if ( source != null ) {
      URL location = source.getLocation();
      System.out.println( qualifiedClassName + " : " + location );
   }
   else {
      System.out.println( qualifiedClassName + " : "
          + "unknown source, likely rt.jar" );
   }
}
 catch ( Exception e ) {
   System.err.println( "Unable to locate class on command line." );
   e.printStackTrace();
}
```

The System class contains several useful class fields and methods. Accessing the current system properties is one of facilities provided by the System class. For example, to get the operating system name string in Java use the static System.getProperty("os.name") method. Here is a sample code to walk through system properties in Java:

```
public class SystemPropertiesWalker  {

   public static void main(String[] args) {
      Properties properties = System.getProperties();
      Set<Object>  sysPropertiesKeys = properties.keySet();
```

```
      for (Object key : sysPropertiesKeys) {
         System.out.println(key + " =" + properties.getProperty((String)key));
      }
   }
}
```

If there is no current set of system properties, a set of system properties is first created and initialized. This set of system properties always includes values for the following keys:

Key	Description of Associated Value
`java.version`	Java Runtime Environment version
`java.vendor`	Java Runtime Environment vendor
`java.vendor.url`	Java vendor URL
`java.home`	Java installation directory
`java.vm.specification.version`	Java Virtual Machine specification version
`java.vm.specification.vendor`	Java Virtual Machine specification vendor
`java.vm.specification.name`	Java Virtual Machine specification name
`java.vm.version`	Java Virtual Machine implementation version
`java.vm.vendor`	Java Virtual Machine implementation vendor
`java.vm.name`	Java Virtual Machine implementation name
`java.specification.version`	Java Runtime Environment specification version
`java.specification.vendor`	Java Runtime Environment specification vendor
`java.specification.name`	Java Runtime Environment specification name
`java.class.version`	Java class format version number
`java.class.path`	Java class path
`java.library.path`	List of paths to search when loading libraries
`java.io.tmpdir`	Default temp file path
`java.compiler`	Name of JIT compiler to use
`java.ext.dirs`	Path of extension directory or directories

os.name	Operating system name
os.arch	Operating system architecture
os.version	Operating system version
file.separator	File separator ("/" on UNIX)
path.separator	Path separator (":" on UNIX)
line.separator	Line separator ("\n" on UNIX)
user.name	User's account name
user.home	User's home directory
user.dir	User's current working directory

We get a compiler error when we declare a variable in a for/while loop without braces. For example,

```
public class XyzwsLoopDemo {

   public static void main(String[] args) {
      for (int x=0; x!=10; x++)
         String str = "LOOP";  //Compiler Error
   }
}
```

But it is fine when you add braces,

```
public class XyzwsLoopDemo {

   public static void main(String[] args) {
      for (int x=0; x!=10; x++) {
         String str = "LOOP";
      }
   }


}
```

Let's look the definition of ForStatement in Java Language Specification:

```
BasicForStatement:
        for ( ForInitopt ; Expressionopt ; ForUpdateopt ) Statement

ForStatementNoShortIf:
        for ( ForInitopt ; Expressionopt ; ForUpdateopt )
         StatementNoShortIf

ForInit:
        StatementExpressionList
        LocalVariableDeclaration

ForUpdate:
        StatementExpressionList

StatementExpressionList:
        StatementExpression
        StatementExpressionList , StatementExpression
```

The body of a ForStatement is a single Statement. Here is definition of Statement in Java Language Specification:

```
Statement:
        StatementWithoutTrailingSubstatement
        LabeledStatement
        IfThenStatement
        IfThenElseStatement
        WhileStatement
        ForStatement

StatementWithoutTrailingSubstatement:
        Block
        EmptyStatement
        ExpressionStatement
        AssertStatement
        SwitchStatement
        DoStatement
        BreakStatement
```

```
        ContinueStatement
        ReturnStatement
        SynchronizedStatement
        ThrowStatement
        TryStatement

StatementNoShortIf:
        StatementWithoutTrailingSubstatement
        LabeledStatementNoShortIf
        IfThenElseStatementNoShortIf
        WhileStatementNoShortIf
        ForStatementNoShortIf
```

The ForStatement expects a statement in its body, but the **String str = "LOOP";** is a LocalVariableDeclarationStatement and is not a valid statement. The LocalVariableDeclarationStatement can appear in a block, a block of code with braces around it is, indeed, a "compound statement". Here is Block definition in JLS:

```
Block:
        { BlockStatementsopt }

BlockStatements:
        BlockStatement
        BlockStatements BlockStatement

BlockStatement:
        LocalVariableDeclarationStatement
        ClassDeclaration
        Statement
```

Every local variable declaration statement is immediately contained by a block. Local variable declaration statements may be intermixed freely with other kinds of statements in the block.

There are few ways to read input string from your console/keyboard. The

following smaple code shows how to read a string from the console/keyboard by using Java.

```
public class ConsoleReadingDemo {

   public static void main(String[] args) {

      // ====
      BufferedReader reader = new BufferedReader(new
InputStreamReader(System.in));
      System.out.print("Please enter user name : ");
      String username = null;
      try {
         username = reader.readLine();
      } catch (IOException e) {
         e.printStackTrace();
      }
      System.out.println("You entered : " + username);

      // ===== In Java 5, Java.util,Scanner is used for this purpose.
      Scanner in = new Scanner(System.in);
      System.out.print("Please enter user name : ");
      username = in.nextLine();
      System.out.println("You entered : " + username);



      // ====== Java 6
      Console console = System.console();
      username = console.readLine("Please enter user name : ");
      System.out.println("You entered : " + username);

   }
```

```
}
```

The last part of code used java.io.Console class. you can not get Console instance from System.Console() when running the demo code through Eclipse. Because eclipse runs your application as a background process and not as a top-level process with a system console.

Java makes a distinction between a user thread and another type of thread known as a daemon thread. The daemon threads are typically used to perform services for user threads. The main() method of the application thread is a user thread. Threads created by a user thread are user thread. JVM doesn't terminates unless all the user thread terminate.

You can explicitly specify a thread created by a user thread to be a daemon thread by calling setDaemon(true) on a Thread object. For example, the clock handler thread, the idle thread, the garbage collector thread, the screen updater thread, and the garbage collector thread are all daemon threads. A new created thread inherits the "daemon-status" of the thread that created it unless you explicitly calling setDaemon on that Thread object to change its status.

Note that the setDaemon() method must be called before the thread's start() method is invoked.Once a thread has started executing (i.e., its start() method has been called) its daemon status cannot be changed. To determine if a thread is a daemon thread, use the accessor method isDaemon().

The difference between these two types of threads is straightforward: If the Java runtime determines that the only threads running in an application are daemon threads (i.e., there are no user threads in existence) the Java runtime promptly closes down the application, effectively stopping all daemon threads dead in their tracks. In order for an application to continue running, it must always have at least one live user thread. In all other respects the Java

runtime treats daemon threads and user threads in exactly the same manner.

The class variables **must** have the keyword ***static*** as the modifier and are declared **inside** a *class* body but **outside** of any *method* bodies.

Only one copy of class variables are created from the point the class is loaded into the JVM until the the class is unloaded, regardless of the **number of object instances** that are created from that class.

If one object instance of that class changes the value of a class variable then all the other instances see the **same** changed value.

You access a class variable directly through the class or through an object instance. Accessing a class variable through the class by specifying **the name of the class, followed by a dot, followed by the name of the variable**.

Class variables can be initialized by an **assignment statement**. Unassigned declaration of class variables will get the **default value** as instance variables.

Class variables are **not** serialized. Static method **can access** static variables and **can't access** instance variables.

There are two excpetions:

Class variable has "**static final**" modifiers which must be assigned a value at the **declaration time** or **inside the static initializer**. You can only **assign one time** and can not reassign its value late.

Class variables can be declared **with or without** the keyword static within an **interface declaration**. Every field declaration in the body of an interface is implicitly *public*, *static*, and *final*. It is permitted to redundantly

specify any or all of these modifiers for such fields.

In the previous FAQ: Which replace function works with regex?, We compare the different replace methods in the String class. Here we will focus on replacing backslashes inside a String.

The path in Windows uses the backslashes between folder names. For example, the path of my java source is "c:\dev\app\source". Since backslash is a special character in Java String, if we want to put this path inside a Java String literal, we need to escape the backslashes, it will be

```
String a1 = "c:\\dev\\app\\source";

System.out.println(a1);  // the output is “c:\dev\app\source”.
```

Now we want to change all the single backslashes to double backslashes, one may think this would work:

```
String b1 = a1.replaceAll("\\", "\\\\");
```

But instead you will get Runtime Exception:

```
java.util.regex.PatternSyntaxException: Unexpected internal error near index
1

\

 ^
```

Note the definition of the replaceAll() method in the String class is:

public String replaceAll(String regex, String replacement)

This method "replaces each substring of this string that matches the given regular expression with the given replacement."

We are dealing with regular expression pattern in Java String literal here.

For a backslash \
The pattern to match that would be \\
The String literal to create that pattern would have to have one backslash to escape each of the two backslashes: \\\\

Yes, four backslashes are for one!

So the correct way to do use the replaceAll() method to double up the backslashes are:

String c1 = a1.replaceAll("\\\\", "\\\\\\\\"); //output is "c:\\dev\\app\\source"

If you want to avoid the complex of regex String literals, there is another option, use the replace method in the String class:

public String replace(char oldChar, char newChar)

This method "returns a new string resulting from replacing all occurrences of oldChar in this string with newChar."

This method deals with the chars, not regex, so:

```
String d1 = a1.replace(“\\”, “\\\\”);   // is “c:\\dev\\app\\source”
```

Put all these together:

```
Public class Test {
   public static void main (String … argv) {

    String a1 = "c:\\dev\\app\\";

   System.out.println(a1);  // the output is “c:\dev\app\source”.

    //String b1 = a1.replaceAll("\\", "\\\\");  //runtime exception

   String c1 = a1.replaceAll(“\\\\”, “\\\\\\\\”);

   System.out.println(c1); // is “c:\\dev\\app\\source”

   String d1 = a1.replace(“\\”, “\\\\”);

   System.out.println(d1); // is “c:\\dev\\app\\source”

   }

}
```

Instance variables are any variables, without "static" field modifier, that are defined within the class body and outside any class's methods body.

Instance variables are in scope as long as their enclosing object is in scope. An instance variable is a part of the object that contains it and cannot live

independently of it.

All object instances have their own copies of instance variables. One object instance can change values of its instance variables without affecting all other instances.

Instance variables can be used by all methods of a class unless the methods are marked with "static" modifier. You access instance variables directly from their containing object instances.

Local variables have the most limited scope. Such a variable is accessible only from the function or block in which it is declared. The local variable's scope is from the line they are declared on until the closing curly brace of the method or code block within which they are declared.

Every local variable declaration statement is immediately contained by a block ({ ... }). Local variable declaration statements may be intermixed freely with other kinds of statements in the block.

A local variable declaration can also appear in the header of a for statement. In this case it is executed in the same manner as if it were part of a local variable declaration statement.

Local variables declaration have one and only one ***final*** modifer can be used.

Local variables are not given default initial values. They **must be** initialized explicitly **before** they are used.

The scope of a variable determines where it can be used. Syntactically, scope of any variable is the code between the curly braces of where it is declared.

An object becomes eligible for garbage collection when it becomes unreachable by any code. Two way can make this happened:

- Explicitly set the reference variable that refers to the object to *null*.
- Reassign the reference variable that points to the object to refer to other object.

For example,

```
class Program {
  public static void main(String[] args) {
    X x1 = new X("1");
    X x2 = new X("2");
    x1 = null;    // X("1") object is eligible for collection after this
    x2 = new Y(); // X("2") object is eligible for collection after this
  }
}
```

The **java.util.Arrays** class contains various methods for manipulating arrays (such as sorting and searching). This class also contains a static factory that allows arrays to be viewed as lists. For example,

```
Long[]  nameIDs = { … … };

List nameIDList = Arrays.asList(nameIDs);
```

You need to add/delete an item in an ArrayList when you are iterating the list. You will receive the java.util.ConcurrentModificationException exception. For example, the following code will throw an exception after adding an item into list:

public class Sample {

public static void main(String[] args) {

List<Integer> iList = new ArrayList<Integer>();

```
  for (int i = 0; i != 100; i++)
    iList.add(i);

  int addValue = 1000;
  for (Integer i: iList) {
    if (i%10 == 0) {
      iList.add(addValue++);
    }
  }

}
```

To avoid java.util.ConcurrentModificationException exception, we can add an item through the iterator of list. If we do the same as the above code, the next access item in list via the iterator will generate the same exception.

```
public class Sample {

  public static void main(String[] args) {

    List<Integer>  iList = new ArrayList<Integer>();
    for (int i = 0; i != 100; i++)
      iList.add(i);

    int addValue = 1000;

    for (ListIterator<Integer> itr = iList.listIterator(); itr.hasNext();) {
      Integer i = itr.next();
      if (i%10 == 0) {
        itr.add(addValue++);
      }
    }
```

```
}
```

Using Map in java is a convenient way to store objects into a collection and identified by their names. The Map interface provides API of adding and getting a particular object by the name it's stored with. However, there is no simple API to loop through all the items in the Map.

An older fashion to do this is to use the java.util.Iterator, for example:

```
public void test () {
    Map testMap = new HashMap();
    testMap.put("key1", "value1");
    testMap.put("key2", "value2");
    Iterator it = testMap.keySet() .iterator () ;
     while (  it.hasNext ()   )   {
        String key =  (String) it.next () ;
        System.out.println ( "key:"  +  key);
        System.out.println ("value:" + testMap.get ( key )  ) ;
    }
  }
```

If you want to take advantage of the

```
  for (Type x: collections) {
```

```
...

}
```

You can by pass the iterator, instead to use the java.util.Map.Entry. The Map.Entry is a name-value pair. The entrySet() method of the Map will return a collection f the the Map.Entry. The following is an example:

```
public void test () {

    Map testMap = new HashMap();

    testMap.put("key1", "value1");

    testMap.put("key2", "value2");

    for (Map.Entry entry: testMap.entrySet()) {

        System.out.println ( "key:"  +  entry.getKey());

        System.out.println ("value:" + entry.getValue() ) ;

    }

}
```

The second way is useful if you want to do the looping inside a JSP with tags:

```
<table>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<c:forEach var="entry" items="${testMap.entrySet()}"
   <tr>
            <td>${entry.key}</td>
```

```
            <td>${entry.value}</td>
</tr>
</c:forEach>
</table>
```

The remainder operator % returns the remainder from the division of the left operand by the right operand;the left-hand operand is the dividend and the right-hand operand is the divisor.
In Java, the remainder operator % can be used with integral operands as well as floating-point operands (§15.17.3 Remainder Operator %).

For the integral type remainder operation whose operands are integers after binary numeric promotion, the remainder operator is defined by the following identity (if b is not 0): (a/b)*b + a%b = a

- If the divisor b evaluates to zero, then an ArithmeticException is thrown.
- If the dividend evaluates to zero, the result is zero.
- The sign of the result equals the sign of the dividend.
- The magnitude of the result |a%b| is always less than the magnitude of the divisor |b|.

The result of a floating-point remainder operation is determined by the rules of IEEE arithmetic:

- If either operand is NaN, the result is NaN.
- If the result is not NaN, the sign of the result equals the sign of the dividend.
- If the dividend is an infinity, or the divisor is a zero, or both, the result is NaN.
- If the dividend is finite and the divisor is an infinity, the result equals the dividend.
- If the dividend is a zero and the divisor is finite, the result equals the

dividend.

- In the remaining cases, where neither an infinity, nor a zero, nor NaN is involved, the floating-point remainder r from the division of a dividend n by a divisor d is defined by the mathematical relation r=n-(d*q) where q is an integer that is negative only if n/d is negative and positive only if n/d is positive, and whose magnitude is as large as possible without exceeding the magnitude of the true mathematical quotient of n and d.

The greedy, reluctant, and possessive quantifiers are for matching a specified expression x number of times. Quantifiers allow you to specify x number of occurrences to match against. The greedy quantifier is used to match with the longest possible string that matches the pattern while the reluctant quantifier is used to match with the shortest possible string that matches the pattern. The possessive quantifier is used to match the regular expression to the entire string and only matches when the whole string satisfies the criteria.

Quantifiers			Meaning
Greedy	Reluctant	Possessive	
`X?`	`X??`	`X?+`	*X*, once or not at all
`X*`	`X*?`	`X*+`	*X*, zero or more times
`X+`	`X+?`	`X++`	*X*, one or more times
`X{n}`	`X{n}?`	`X{n}+`	*X*, exactly *n* times
`X{n,}`	`X{n,}?`	`X{n,}+`	*X*, at least *n* times
`X{n,m}`	`X{n,m}?`	`X{n,m}+`	*X*, at least *n* but not more than *m* times

For example we have string: **xxfooooooooofoo**, let's see what the differences among greedy, reluctant, and possessive qualifiers are in the matching strategy:

Given the regex **(\w)*(.foo)**, the first part **(\w)*** is greedy and the entired string is consumed by it. At this point, the second part **(.foo)** cannot be

matched (no portion of string left) and the overall expression cannot succeed. So the matcher slowly backs off one letter at a time until the rightmost occurrence of **ofoo** (which matches the second part **.foo**) has been regurgitated, at which point the match succeeds and the search ends. So the matched string is **xxfooooooooofoo**.

Given the regex **(\w)*?(.foo)**, the fist part **(\w)*?** is reluctant, so it starts by first consuming "nothing". Because the first four characters do not match **.foo**, the matcher is forced to swallow the first letter (an "**x**"), which triggers the first match **xxfoo** string. The matcher will continues the process until the entire string is exhausted. The matcher will find another match string **ooooooofoo**.

Given the regex **(\w)*+(.foo)**, the first part **(\w)*+** is possessive and the entired string is consumed by it, there is nothing left for the second part **.foo** and the overall pattern fails. Unlike greedy match, there are no steps to backtrack to for possessive match. The match attempt fails immediately when the second **.foo** matching fails. If we modify the given string to **xxfooooooooo=foo**, the first part grabs **xxfooooooooo** and the second part **.foo** will match the left string **=foo**. So the match result will be **xxfooooooooo=foo**. Use a possessive quantifier for situations where you want to seize all of something without ever backing off; it will outperform the equivalent greedy quantifier in cases where the match is not immediately found.