Using Database Operations

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 820–4992
June 2008

080906@20795

Contents

CHAPTER 1

Using Database Operations

The following sections provide instructions on how to use database operations when designing a project. The following database adapters are covered:

- "Using DB2 Operations" on page 51
- "Using DB2 Connect Operations" on page 62
- "Using JDBC/ODBC Operations" on page 5
- "Using Oracle Operations" on page 76
- "Using SQL ServerOperations" on page 37
- "Using Sybase Operations" on page 18
- "Using VSAM Operations" on page 30

Using JDBC/ODBC Operations

Database operations in the JDBC Adapter are used to access the JDBC database. Database operations are either accessed through Activities in BPEL, or through methods called from a JCD Collaboration.

- "JDBC Adapter Database Operations (BPEL)" on page 5
- "JDBC Adapter Database Operations (JCD)" on page 7

JDBC Adapter Database Operations (BPEL)

Within a BPEL business process, the JDBC Adapter uses BPEL Activities to perform basic outbound database operations, including:

- Insert
- Update
- Delete
- SelectOne
- SelectMultiple

- SelectAll

In addition to these outbound operations, the JDBC Adapter also employs the inbound Activity **ReceiveOne** within a Prepared Statement OTD.

The ability to perform any of the above methods using a table OTD may not be possible with all third-party drivers. You have to use a Prepared Statement to perform such an operation. Check with the respective driver's vendor for further information. This feature is known as Updatable ResultSet.

Activity Input and Output

The Sun SeeBeyond Enterprise Designer– Business Rules Designer includes Input and Output columns to map and transform data between Activities displayed on the Business Process Canvas.

Figure 1–1 displays the business rules between the **FileClient.write** and **otdJDBC.Db_employeeDelete** Activities. In this example, the **whereClause** appears on the Input side.

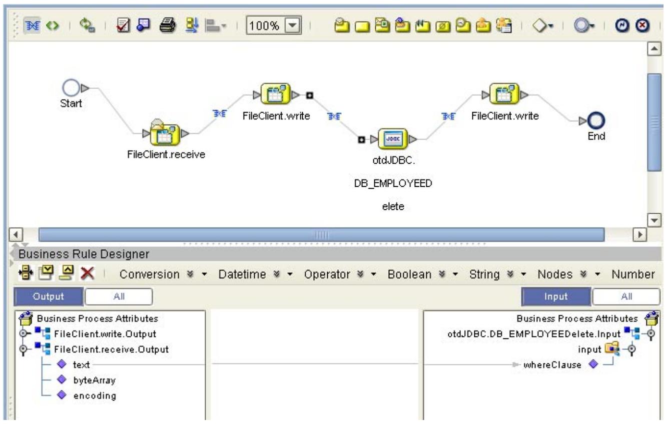


FIGURE 1–1 Input and Output Between Activities

The following table lists the expected Input and Output of each database operation Activity.

TABLE 1–1 JDBC/ODBC Operations

eInsight Operations	Activity Input	Activity Output
SelectAll	where() clause (optional)	Returns all rows that fit the condition of the where() clause.
SelectMultiple	number of rows where() clause (optional)	Returns the number of rows specified that fit the condition of the where() clause, and the number of rows to be returned. For example: If the number of rows that meet the condition are 5 and the number of available rows are 10, then only 5 rows will be returned. Alternately, if the number of rows that meet the condition are 20, but if the number of available rows are 10, then only 10 rows are returned.
SelectOne	where() clause (optional)	Returns the first row that fits the condition of the where() clause.
Insert	definition of new item to be inserted	Returns status.
Update	where() clause	Returns status.
Delete	where() clause	Returns status.

JDBC Adapter Database Operations (JCD)

The same database operations are also used in the JCD, but appear as methods to call from the Collaboration.

Tables, Views, and Stored Procedures are manipulated through OTDs. Methods to call include:

- insert()
- insertRow()
- update(*String sWhere*)
- updateRow()
- delete(*String sWhere*)
- deleteRow()
- select(*String where*)

The ability to perform any of the above methods using a table OTD may not be possible with all third-party drivers. You have to use a Prepared Statement to perform such an operation. Check with the respective driver's vendor for further information. This feature is known as Updatable ResultSet.

Note – Refer to the Javadoc for a full description of methods included in the JDBC Adapter.

The Table

A table OTD represents a database table. It consists of fields and methods. Fields correspond to the columns of a table while methods are the operations that you can apply to the OTD. This allows you to perform Query, Update, Insert, and Delete SQL operations in a table. The ability to update via a ResultSet is called "Updatable ResultSet", which is a feature supported by this adapter.

By default, the Table OTD has UpdatableConcurrency and ScrollTypeForwardOnly. Normally you do not have to change the default setting.

The type of result returned by the select() method can be specified using:

- SetConcurrencytoUpdatable
- SetConcurrencytoReadOnly
- SetScrollTypetoForwardOnly
- SetScrollTypetoScrollSensitive
- SetScrollTypetoInsensitive

The Query (Select) Operation

▼ To perform a query operation on a table:

1. **Execute the select() method with the "where" clause specified if necessary.**

2. **Loop through the ResultSet using the next() method.**

3. **Process the return record within a while() loop.**

   For example:

```
package prjJDBC_JCDjcdALL;

public class jcdTableSelect
{

    public com.stc.codegen.logger.Logger logger;
    public com.stc.codegen.alerter.Alerter alerter;

    public com.stc.codegen.util.CollaborationContext collabContext;

    public com.stc.codegen.util.TypeConverter typeConverter;

    public void receive( com.stc.connector.appconn.file.FileTextMessage
 input, dtd.otdInputDTD_1394195520.DBemployees otdInputDTD_DBemployees_1,
 otdJDBC.OtdJDBCOTD otdJDBC_1, dtd.otdOutputDTD882991309.DBemployee
 otdOutputDTD_DBemployee_1, com.stc.connector.appconn.file.FileApplication
 FileClient_1 ) throws Throwable
    {
        FileClient_1.setText( "Selecting record(s) from db_employee table via
 table select .." );
        FileClient_1.write();
        otdJDBC_1.getDB_EMPLOYEE().select( input.getText() );
        while (otdJDBC_1.getDB_EMPLOYEE().next()) {
            otdOutputDTD_DBemployee_1.setEmpNo( typeConverter.shortToString(
 otdJDBC_1.getDB_EMPLOYEE().getEMP_NO(), "#", false, "" ) );
            otdOutputDTD_DBemployee_1.setLastname(
 otdJDBC_1.getDB_EMPLOYEE().getLAST_NAME() );
            otdOutputDTD_DBemployee_1.setFirstname(
 otdJDBC_1.getDB_EMPLOYEE().getFIRST_NAME() );
            otdOutputDTD_DBemployee_1.setRate(
 otdJDBC_1.getDB_EMPLOYEE().getRATE().toString() );
            otdOutputDTD_DBemployee_1.setLastDate(
 typeConverter.dateToString( otdJDBC_1.getDB_EMPLOYEE().getLAST_UPDATE(),
 "yyyy-MM-dd hh:mm:ss", false, "" ) );
            FileClient_1.setText( otdOutputDTD_DBemployee_1.marshalToString() );
            FileClient_1.write();
        }
        FileClient_1.setText( "Done table select." );
        FileClient_1.write();
    }

}
```

The Insert Operation

▼ To perform an insert operation on a table:

1. **Execute the insert() method. Assign a value to a field.**
2. **Insert the row by calling insertRow()**

   This example inserts an employee record.

```
package prjJDBC_JCDjcdALL;

public class jcdInsert
{

    public com.stc.codegen.logger.Logger logger;

    public com.stc.codegen.alerter.Alerter alerter;

    public com.stc.codegen.util.CollaborationContext collabContext;

    public com.stc.codegen.util.TypeConverter typeConverter;

    public void receive( com.stc.connector.appconn.file.FileTextMessage input,
 com.stc.connector.appconn.file.FileApplication FileClient_1,
dtd.otdInputDTD_1394195520.DBemployees otdInputDTD_DBemployees_1,
dtd.otdOutputDTD882991309.DBemployee otdOutputDTD_DBemployee_1,
otdJDBC.OtdJDBCOTD otdJDBC_1 )
        throws Throwable
    {
        FileClient_1.setText( "Inserting records into db_employee table .." );
        FileClient_1.write();
        otdInputDTD_DBemployees_1.unmarshalFromString( input.getText() );
        for (int i1 = 0; i1 < otdInputDTD_DBemployees_1.countX_sequence_A(); i1 += 1) {
            otdJDBC_1.getInsert_Ps().setEmp_no( typeConverter.stringToShort(
otdInputDTD_DBemployees_1.getX_sequence_A( i1 ).getEmpNo(), "#", false, 0 ) );
            otdJDBC_1.getInsert_Ps().setLast_name(
otdInputDTD_DBemployees_1.getX_sequence_A( i1 ).getLastname() );
            otdJDBC_1.getInsert_Ps().setFirst_name(
otdInputDTD_DBemployees_1.getX_sequence_A( i1 ).getFirstname() );
            otdJDBC_1.getInsert_Ps().setRate(
new java.math.BigDecimal( otdInputDTD_DBemployees_1.getX_sequence_A( i1 ).
getRate() ) );            otdJDBC_1.getInsert_Ps().setLast_update(
typeConverter.stringToSQLDate( otdInputDTD_DBemployees_1.
getX_sequence_A( i1 ).getLastDate(), "yyyy-MM-dd hh:mm:ss", false, "" ) );
            otdJDBC_1.getInsert_Ps().executeUpdate();
        }
        FileClient_1.setText( "Done Insert." );
        FileClient_1.write();
    }

}
```

The Update Operation

▼ To perform an update operation on a table:

1. **Execute the update() method.**
2. **Using a while loop together with next(), move to the row that you want to update.**
3. **Assign updating value(s) to the fields of the table OTD**
4. **Update the row by calling updateRow().**

```
package prjJDBC_JCDjcdALL;

public class jcdUpdate
{

    public com.stc.codegen.logger.Logger logger;

    public com.stc.codegen.alerter.Alerter alerter;

    public com.stc.codegen.util.CollaborationContext collabContext;

    public com.stc.codegen.util.TypeConverter typeConverter;

    public void receive( com.stc.connector.appconn.file.FileTextMessage input,
otdJDBC.OtdJDBCOTD otdJDBC_1, dtd.otdOutputDTD882991309.DBemployee
otdOutputDTD_DBemployee_1, dtd.otdInputDTD_1394195520.DBemployees
otdInputDTD_DBemployees_1, com.stc.connector.appconn.file.FileApplication
FileClient_1 ) throws Throwable
    {
        FileClient_1.setText( "Update the Rate and Last_update
 fields using Prepared Statement.. " );
        FileClient_1.write();
        otdJDBC_1.getUpdate_Ps().setEmp_no( typeConverter.stringToShort(
 input.getText(), "#", false, 0 ) );
        otdJDBC_1.getUpdate_Ps().executeUpdate();
        FileClient_1.setText( "Done Update." );
        FileClient_1.write();
    }

}
```

The Delete Operation

▼ To perform a delete operation on a table:

- **Execute the delete() method.**

  In this example DELETE an employee.

```
package prjJDBC_JCDjcdALL;

public class jcdDelete
{

    public com.stc.codegen.logger.Logger logger;

    public com.stc.codegen.alerter.Alerter alerter;

    public com.stc.codegen.util.CollaborationContext collabContext;

    public com.stc.codegen.util.TypeConverter typeConverter;

    public void receive( com.stc.connector.appconn.file.
FileTextMessage input, dtd.otdInputDTD_1394195520.DBemployees
 otdInputDTD_DBemployees_1, otdJDBC.OtdJDBCOTD otdJDBC_1,
 dtd.otdOutputDTD882991309.DBemployee otdOutputDTD_DBemployee_1,
 com.stc.connector.appconn.file.FileApplication FileClient_1 )
        throws Throwable
    {
        FileClient_1.setText( "Delete record .." );
        FileClient_1.write();
        otdJDBC_1.getDB_EMPLOYEE().delete( input.getText() );
        FileClient_1.setText( "Done delete." );
        FileClient_1.write();
    }

}
```

The Stored Procedure

A Stored Procedure OTD represents a database stored procedure. Fields correspond to the arguments of a stored procedure while methods are the operations that you can apply to the OTD. It allows you to execute a stored procedure. Remember that while in the Collaboration Editor you can drag and drop nodes from the OTD into the Collaboration Editor.

Executing Stored Procedures

The OTD represents the Stored Procedure "LookUpGlobal" with two parameters, an inbound parameter (INLOCALID) and an outbound parameter (OUTGLOBALPRODUCTID). These inbound and outbound parameters are generated by the Database Wizard and are represented in the resulting OTD as nodes. Within the Transformation Designer, you can drag values from the input parameters, execute the call, collect data, and drag the values to the output parameters.

▼ Steps for executing the Stored Procedure include:

1. **Specify the input values.**
2. **Execute the Stored Procedure.**
3. **Retrieve the output parameters if any.**

   For example:

```
package Storedprocedure;

public class sp_jce
{

    public com.stc.codegen.logger.Logger logger;

    public com.stc.codegen.alerter.Alerter alerter;

    public void receive( com.stc.connector.appconn.file.FileTextMessage
 input,com.stc.connector.appconn.file.FileApplication FileClient_1,
employeedb.Db_employee employeedb_with_top_db_employee_1,
insert_DB.Insert_DBOTD insert_DB_1 )
    throws Throwable
    {
employeedb_with_top_db_employee_1.unmarshalFromString( input.getText() );

insert_DB_1.getInsert_new_employee().setEmployee_no(
 java.lang.Integer.parseInt( employeedb_with_top_db_employee_1.
getEmployee_no() ) );

insert_DB_1.getInsert_new_employee().setEmployee_Lname(
 employeedb_with_top_db_employee_1.getEmployee_lname() );

insert_DB_1.getInsert_new_employee().setEmployee_Fname(
 employeedb_with_top_db_employee_1.getEmployee_fname() );

insert_DB_1.getInsert_new_employee().setRate(
 java.lang.Float.parseFloat( employeedb_with_top_db_employee_1.
getRate() ) );

insert_DB_1.getInsert_new_employee().setUpdate_date(
 java.sql.Timestamp.valueOf( employeedb_with_top_db_employee_1.
getUpdate_date() ) );

insert_DB_1.getInsert_new_employee().execute();

insert_DB_1.commit();

FileClient_1.setText( "procedure executed" );

FileClient_1.write();
    }

}
```

Manipulating the ResultSet and Update Count Returned by Stored Procedure

For Stored Procedures that return ResultSets and Update Count, the following methods are provided to manipulate the ResultSet:

- enableResultSetOnly
- enableUpdateCountsOnly
- enableResultSetandUpdateCounts
- resultsAvailable
- next
- getUpdateCount
- available

Many drivers do not support manipulating ResultSets in a Stored Procedure. It is recommended that you use specific Adapters for Oracle, SQL Server, Sybase, DB2, and so forth, to peform such operations.

JDBC stored procedures do not return records as ResultSets. Instead, the records are returned through output reference cursor parameters. Reference Cursor parameters are essentially ResultSets.

The **resultsAvailable()** method, added to the PreparedStatementAgent class, simplifies the whole process of determining whether any results, be it Update Counts or ResultSets, are available after a stored procedure has been executed. Although JDBC provides three methods (**getMoreResults()**, **getUpdateCount()**, and **getResultSet()**) to access the results of a stored procedure call, the information returned from these methods can be quite confusing to the inexperienced Java JDBC programmer and they also differ between vendors. You can simply call **resultsAvailable()** and if Boolean true is returned, you can expect either a valid Update

Count when **getUpdateCount()** is called and/or the next ResultSet has been retrieved and made available to one of the ResultSet nodes defined for the Stored Procedure OTD, when that node's **available()** method returns true.

Frequently, Update Counts information that is returned from a Stored Procedures is insignificant. You should process returned ResultSet information and avoid looping through all of the Update Counts. The following three methods control exactly what information should be returned from a stored procedure call. The **enableResultSetsOnly()** method, added to the PreparedStatement Agent class allows only ResultSets to be returned and thus every **resultsAvailable()** called only returns Boolean true if a ResultSet is available. Likewise, the **enableUpdateCountsOnly()** causes **resultsAvailable()** to return true only if an Update Count is available. The default case of **enableResultsetsAndUpdateCount()** method allows both ResultSets and Update Counts to be returned.

Collaboration usability for a stored procedure ResultSet

The Column data of the ResultSets can be dragged-and-dropped from their nodes to the Business Rules. Below is a code snippet that can be generated by the Collaboration Editor:

```
while (getSPIn().getSpS_multi().resultsAvailable())
{
if (getSPIn().getSpS_multi().getUpdateCount() > 0)
  {
    System.err.println("Updated "+getSPIn().getSpS_multi().getUpdateCount()+" rows");
  }

  if (getSPIn().getSpS_multi().getNormRS().available())
  {
    while (getSPIn().getSpS_multi().getNormRS().next())
    {
      System.err.println("Customer Id   = "+getSPIn().
getSpS_multi().getNormRS().getCustomerId());
      System.err.println("Customer Name = "+getSPIn().
getSpS_multi().getNormRS().getCustomerName());
      System.err.println();
    }
    System.err.println("===");
  }
  else if (getSPIn().getSpS_multi().getDbEmployee().available())
  {
    while (getSPIn().getSpS_multi().getDbEmployee().next())
    {
      System.err.println("EMPNO    = "+getSPIn().getSpS_multi().
getDbEmployee().getEMPNO());
      System.err.println("ENAME    = "+getSPIn().getSpS_multi().
getDbEmployee().getENAME());
      System.err.println("JOB      = "+getSPIn().getSpS_multi().
getDbEmployee().getJOB());
      System.err.println("MGR      = "+getSPIn().getSpS_multi().
getDbEmployee().getMGR());
      System.err.println("HIREDATE = "+getSPIn().getSpS_multi().
getDbEmployee().getHIREDATE());
      System.err.println("SAL      = "+getSPIn().getSpS_multi().
getDbEmployee().getSAL());
      System.err.println("COMM     = "+getSPIn().getSpS_multi().
getDbEmployee().getCOMM());
      System.err.println("DEPTNO   = "+getSPIn().getSpS_multi().
getDbEmployee().getDEPTNO());
      System.err.println();
    }
    System.err.println("===");
  }
}
```

Note – resultsAvailable() and **available()** cannot be indiscriminately called because each time they move ResultSet pointers to the appropriate locations.

After calling "**resultsAvailable()**", the next result (if available) can be either a **ResultSet** or an **UpdateCount** if the default "**enableResultSetsAndUpdateCount()**" was used.

Because of limitations imposed by some DBMSs, it is recommended that for maximum portability, all of the results in a ResultSet object should be retrieved before OUT parameters are retrieved. Therefore, you should retrieve all ResultSet(s) and Update Counts first followed by retrieving the OUT type parameters and return values.

The following list includes specific ResultSet behavior that you may encounter:

- The method **resultsAvailable()** implicitly calls **getMoreResults()** when it is called more than once. You should not call both methods in your java code. Doing so may result in skipped data from one of the ResultSets when more than one ResultSet is present.
- The methods **available()** and **getResultSet()** can not be used in conjunction with multiple ResultSets being open at the same time. Attempting to open more the one ResultSet at the same time closes the previous ResultSet. The recommended working pattern is:
  - Open one ResultSet (ResultSet_1) and work with the data until you have completed your modifications and updates. Open ResultSet_2, (ResultSet_1 is now closed) and modify. When you have completed your work in ResultSet_2, open any additional ResultSets or close ResultSet_2.
- If you modify the ResultSet generated by the Execute mode of the Database Wizard, you need to assure the indexes match the stored procedure. By doing this, your ResultSet indexes are preserved.
- Generally, getMoreResults does not need to be called. It is needed if you do not want to use our enhanced methods and you want to follow the traditional JDBC calls on your own.

The DBWizard Assistant expects the column names to be in English when creating a ResultSet.

Prepared Statement

A Prepared Statement OTD represents a SQL statement that has been compiled. Fields in the OTD correspond to the input values that users need to provide.

Prepared statements can be used to perform insert, update, delete and query operations. A prepared statement uses a question mark (?) as a place holder for input. For example:

```
insert into EMP_TAB(Age, Name, Dept No) value(?, ?, ?)
```

To execute a prepared statement, set the input parameters and call **executeUpdate()** and specify the input values if any.

```
getPrepStatement().getPreparedStatementTest().setAge(23);
getPrepStatement().getPreparedStatementTest().setName(”Peter Pan’);
getPrepStatement().getPreparedStatementTest().setDeptNo(6);
getPrepStatement().getPreparedStatementTest().executeUpdate();
```

Note – Drivers must be able to support metadata calls to view column information. Some drivers may not support metadata calls, in which case you must add the columns manually. For drivers that do support metadata calls, prefill the column information. For drivers that do not support the meta data call, column information can be left blank.

Batch Operations

To achieve better performance, consider using a bulk insert if you have to insert many records. This is the “Add Batch” capability. The only modification required is to include the **addBatch()** method for each SQL operation and then the **executeBatch()** call to submit the batch to the database server. Batch operations apply only to Prepared Statements.

Not all drivers support batch operations. Check with the respective driver’s vendor for further information.

```
getPrepStatement().getPreparedStatementTest().setAge(23);
getPrepStatement().getPreparedStatementTest().setName(”Peter Pan’);
getPrepStatement().getPreparedStatementTest().setDeptNo(6);
getPrepStatement().getPreparedStatementTest().addBatch();

getPrepStatement().getPreparedStatementTest().setAge(45);
getPrepStatement().getPreparedStatementTest().setName(”Harrison Ford’);
getPrepStatement().getPreparedStatementTest().setDeptNo(7);
getPrepStatement().getPreparedStatementTest().addBatch();
getPrepStatement().getPreparedStatementTest().executeBatch();
```

Using Sybase Operations

The database operations used in the Sybase Adapter are used to access the Sybase database. Database operations are either accessed through Activities in BPEL, or through methods called from a JCD Collaboration.

- "Sybase Adapter Database Operations (BPEL)" on page 18
- "Sybase Adapter Database Operations (JCD)" on page 20

Sybase Adapter Database Operations (BPEL)

The Sybase Adapter uses a number of operations to query the Sybase database. Within a BPEL business process, the Sybase Adapter uses BPEL Activities to perform basic outbound database operations, including:

- Insert
- Update
- Delete
- SelectOne
- SelectMultiple
- SelectAll

In addition to these outbound operations, the Sybase Adapter also employs the inbound Activity **ReceiveOne** within a Prepared Statement OTD.

Activity Input and Output

The Sun SeeBeyond Enterprise Designer– Business Rules Designer includes Input and Output columns to map and transform data between Activities displayed on the Business Process Canvas.

Figure 1–2 displays the business rules between the **FileClient.write** and **otdSybase.Db_employeeDelete** Activities. In this example, the **whereClause** appears on the Input side.

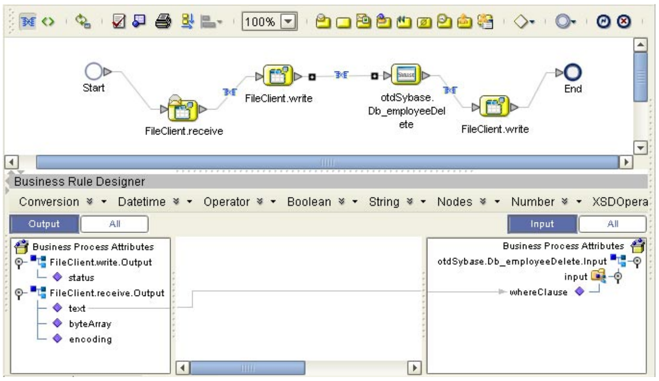


FIGURE 1–2 Input and Output Between Activities

The following table lists the expected Input and Output of each database operation Activity.

TABLE 1–2 Sybase Operations

eInsight Operations	Activity Input	Activity Output
SelectAll	where() clause (optional)	Returns all rows that fit the condition of the where() clause.
SelectMultiple	number of rows wherels() clause (optional)	Returns the number of rows specified that fit the condition of the where() clause, and the number of rows to be returned. For example: If the number of rows that meet the condition are 5 and the number of available rows are 10, then only 5 rows will be returned. Alternately, if the number of rows that meet the condition are 20, but if the number of available rows are 10, then only 10 rows are returned.
SelectOne	where() clause (optional)	Returns the first row that fits the condition of the where() clause.
Insert	definition of new item to be inserted	Returns status.
Update	where() clause	Returns status.
Delete	where() clause	Returns status.

Sybase Adapter Database Operations (JCD)

The same database operations are also used in the JCD, but appear as methods to call from the Collaboration.

Tables, Views, and Stored Procedures are manipulated through OTDs. Methods to call include:

- insert()
- insertRow()
- update(*String sWhere*)
- updateRow()
- delete(*String sWhere*)
- deleteRow()
- select(*String where*)

Note – Refer to the Javadoc for a full description of methods included in the Sybase Adapter.

The Table

A table OTD represents a database table. It consists of fields and methods. Fields correspond to the columns of a table while methods are the operations that you can apply to the OTD. This allows you to perform query, update, insert, and delete SQL operations in a table. The ability to update via a resultset is called "Updatable Resultset", which is a feature supported by this adapter.

By default, the Table OTD has UpdatableConcurrency and ScrollTypeForwardOnly. Normally you do not have to change the default setting.

The type of result returned by the select() method can be specified using:

- SetConcurrencytoUpdatable
- SetConcurrencytoReadOnly
- SetScrollTypetoForwardOnly
- SetScrollTypetoScrollSensitive
- SetScrollTypetoInsensitive

The Query (Select) Operation

▼ To perform a query operation on a table:

1. **Execute the select() method with the "where" clause specified if necessary.**
2. **Loop through the ResultSet using the next() method.**
3. **Process the return record within a while() loop.**

   For example:

```
package prjSybase_JCDjcdALL;


public class jcdTableSelect
{

    public com.stc.codegen.logger.Logger logger;

    public com.stc.codegen.alerter.Alerter alerter;

    public com.stc.codegen.util.CollaborationContext collabContext;

    public com.stc.codegen.util.TypeConverter typeConverter;

    public void receive( com.stc.connector.appconn.file.
FileTextMessage input, dtd.otdOutputDTD1325973702.DB_Employee
 otdOutputDTD_DB_Employee_1, otdSybase.OtdSybaseOTD otdSybase_1,
 com.stc.connector.appconn.file.FileApplication FileClient_1 )
        throws Throwable
    {
        FileClient_1.setText( "Selecting records from db_employee
 table via Table Select........" );
        FileClient_1.write();
        otdSybase_1.getDb_employee().select( input.getText() );
        while (otdSybase_1.getDb_employee().
next()) {
            otdOutputDTD_DB_Employee_1.setEmpNo( typeConverter.
shortToString( otdSybase_1.getDb_employee().getEMP_NO(), "#",
 false, "" ) );
            otdOutputDTD_DB_Employee_1.setLastname(
 otdSybase_1.getDb_employee().getLAST_NAME() );
            otdOutputDTD_DB_Employee_1.setFirstname(
 otdSybase_1.getDb_employee().getFIRST_NAME() );
            otdOutputDTD_DB_Employee_1.setRate(
 otdSybase_1.getDb_employee().getRATE().toString() );
            otdOutputDTD_DB_Employee_1.setLastDate(
 typeConverter.dateToString( otdSybase_1.getDb_employee().
getLAST_UPDATE(), "yyyy-MM-dd hh:mm:ss", false, "" ) );
            FileClient_1.setText( otdOutputDTD_DB_Employee_1.
marshalToString() );
            FileClient_1.write();
        }
        FileClient_1.setText( "Table Select Done." );
        FileClient_1.write();
    }

}
```

The Insert Operation

To perform an insert operation on a table:

1. **Execute the insert() method. Assign a field.**
2. **Insert the row by calling insertRow()**

   This example inserts an employee record.

```
package prjSybase_JCDjcdALL;


public class jcdInsert
{

    public com.stc.codegen.logger.Logger logger;

    public com.stc.codegen.alerter.Alerter alerter;

    public com.stc.codegen.util.CollaborationContext collabContext;

    public com.stc.codegen.util.TypeConverter typeConverter;

    public void receive( com.stc.connector.appconn.file.
FileTextMessage input, otdSybase.OtdSybaseOTD otdSybase_1,
     dtd.otdInputDTD_1206505729.DB_Employee otdInputDTD_DB_Employee_1,
 com.stc.connector.appconn.file.FileApplication FileClient_1 )
        throws Throwable
    {
        FileClient_1.setText( "Inserting records in to
 db_employee table......" );
        FileClient_1.write();
        otdInputDTD_DB_Employee_1.unmarshalFromString(
 input.getText() );
        otdSybase_1.getDb_employee().insert();
        for (int i1 = 0; i1 < otdInputDTD_DB_Employee_1.
countX_sequence_A(); i1 += 1) {
            otdSybase_1.getDb_employee().setEMP_NO(
 typeConverter.stringToShort( otdInputDTD_DB_Employee_1.
getX_sequence_A( i1 ).getEmpNo(), "#", false, 0 ) );
            otdSybase_1.getDb_employee().setLAST_NAME(
 otdInputDTD_DB_Employee_1.getX_sequence_A( i1 ).getLastname() );
            otdSybase_1.getDb_employee().setFIRST_NAME(
 otdInputDTD_DB_Employee_1.getX_sequence_A( i1 ).getFirstname() );
            otdSybase_1.getDb_employee().setRATE(
 new java.math.BigDecimal( otdInputDTD_DB_Employee_1.
getX_sequence_A( i1 ).getRate() ) );
            otdSybase_1.getDb_employee().setLAST_UPDATE(
 typeConverter.stringToTimestamp( otdInputDTD_DB_Employee_1.
getX_sequence_A( i1 ).getLastDate(), "yyyy-MM-dd hh:mm:ss", false, "" ) );
    otdSybase_1.getDb_employee().insertRow();
        }
        FileClient_1.setText( "Insert Done." );
        FileClient_1.write();
    }

}
```

The Update Operation

To perform an update operation on a table:

1. **Execute the update() method.**
2. **Using a while loop together with next(), move to the row that you want to update.**
3. **Assign updating value(s) to the fields of the table OTD**
4. **Update the row by calling updateRow().**

```
package prjSybase_JCDjcdALL;


public class jcdUpdate
{

    public com.stc.codegen.logger.Logger logger;

    public com.stc.codegen.alerter.Alerter alerter;

    public com.stc.codegen.util.CollaborationContext collabContext;
    public com.stc.codegen.util.TypeConverter typeConverter;

    public void receive( com.stc.connector.appconn.file.
FileTextMessage input, otdSybase.OtdSybaseOTD otdSybase_1,
com.stc.connector.appconn.file.FileApplication FileClient_1 )
        throws Throwable
    {
        FileClient_1.setText( "Updating the Rate and Last_update fields .. " );
        FileClient_1.write();
        otdSybase_1.getDb_employee().update( input.getText() );
        while (otdSybase_1.getDb_employee().next()) {
            otdSybase_1.getDb_employee().setLAST_NAME( "Krishna" );
            otdSybase_1.getDb_employee().setFIRST_NAME( "Kishore" );
            otdSybase_1.getDb_employee().updateRow();
        }
        FileClient_1.setText( "Update Done." );
        FileClient_1.write();
    }

}
```

The Delete Operation

▼ To perform a delete operation on a table:

- **Execute the delete() method.**

  In this example DELETE an employee.

```
package prjSybase_JCDjcdALL;

public class jcdDelete
{
public com.stc.codegen.logger.Logger logger;
public com.stc.codegen.alerter.Alerter alerter;
public com.stc.codegen.util.CollaborationContext collabContext;

    public com.stc.codegen.util.TypeConverter typeConverter;

    public void receive( com.stc.connector.appconn.file.
FileTextMessage input, otdSybase.OtdSybaseOTD otdSybase_1,
com.stc.connector.appconn.file.FileApplication FileClient_1 )
        throws Throwable
    {
        FileClient_1.setText( "Deleting record..........." );
        FileClient_1.write();
        otdSybase_1.getDb_employee().delete( input.getText() );
        FileClient_1.setText( "Delete Done." );
        FileClient_1.write();
    }

}
```

The Stored Procedure

A Stored Procedure OTD represents a database stored procedure. Fields correspond to the arguments of a stored procedure while methods are the operations that you can apply to the OTD. It allows you to execute a stored procedure. Remember that while in the Collaboration Editor, you can drag and drop nodes from the OTD into the Collaboration Editor.

Executing Stored Procedures

The OTD represents the Stored Procedure "LookUpGlobal" with two parameters:

- An inbound parameter (INLOCALID)
- An outbound parameter (OUTGLOBALPRODUCTID)

These inbound and outbound parameters are generated by the DataBase Wizard and are represented in the resulting OTD as nodes. Within the Transformation Designer, you can drag values from the input parameters, execute the call, collect data, and drag the values to the output parameters.

▼ Steps for executing the Stored Procedure include:

1. **Specify the input values.**

2. **Execute the Stored Procedure.**

3. **Retrieve the output parameters if any.**

   For example:

```
package Storedprocedure;

public class sp_jce
{

    public com.stc.codegen.logger.Logger logger;

    public com.stc.codegen.alerter.Alerter alerter;

    public void receive( com.stc.connector.appconn.file.FileTextMessage
input,com.stc.connector.appconn.file.FileApplication FileClient_1,
employeedb.Db_employee employeedb_with_top_db_employee_1,insert_DB.
Insert_DBOTD insert_DB_1 )
    throws Throwable
    {
employeedb_with_top_db_employee_1.unmarshalFromString( input.getText() );

insert_DB_1.getInsert_new_employee().setEmployee_no( java.lang.Integer.
parseInt( employeedb_with_top_db_employee_1.getEmployee_no() ) );

insert_DB_1.getInsert_new_employee().setEmployee_Lname(
 employeedb_with_top_db_employee_1.getEmployee_lname() );

insert_DB_1.getInsert_new_employee().setEmployee_Fname(
 employeedb_with_top_db_employee_1.getEmployee_fname() );

insert_DB_1.getInsert_new_employee().setRate(
 java.lang.Float.parseFloat( employeedb_with_top_db_employee_1.getRate() ) );

insert_DB_1.getInsert_new_employee().setUpdate_date(
 java.sql.Timestamp.valueOf( employeedb_with_top_db_employee_1.getUpdate_date() ) );

insert_DB_1.getInsert_new_employee().execute();

insert_DB_1.commit();

FileClient_1.setText( "procedure executed" );

FileClient_1.write();
    }

}
```

Manipulating the ResultSet and Update Count Returned by Stored Procedure

For Stored Procedures that return ResultSets and Update Count, the following methods are provided to manipulate the ResultSet:

- enableResultSetOnly
- enableUpdateCountsOnly
- enableResultSetandUpdateCounts
- resultsAvailable
- next
- getUpdateCount
- available

Sybase stored procedures do not return records as ResultSets, instead, the records are returned through output reference cursor parameters. Reference Cursor parameters are essentially ResultSets.

The **resultsAvailable()** method, added to the PreparedStatementAgent class, simplifies the whole process of determining whether any results, be it Update Counts or ResultSets, are available after a stored procedure has been executed. Although JDBC provides three methods (**getMoreResults()**, **getUpdateCount()**, and **getResultSet()**) to access the results of a stored procedure call, the information returned from these methods can be quite confusing to the inexperienced Java JDBC programmer and they also differ between vendors. You can simply call **resultsAvailable()** and if Boolean true is returned, you can expect either a valid Update Count when **getUpdateCount()** is called and/or the next ResultSet has been retrieved and made available to one of the ResultSet nodes defined for the Stored Procedure OTD, when that node's **available()** method returns true.

Frequently, Update Counts information that is returned from a Stored Procedures is insignificant. You should process returned ResultSet information and avoid looping through all of the Update Counts. The following three methods control exactly what information should be returned from a stored procedure call. The **enableResultSetsOnly()** method, added to the PreparedStatement Agent class allows only ResultSets to be returned and thus every **resultsAvailable()** called only returns Boolean true if a ResultSet is available. Likewise, the **enableUpdateCountsOnly()** causes **resultsAvailable()** to return true only if an Update Count is available. The default case of **enableResultsetsAndUpdateCount()** method allows both ResultSets and Update Counts to be returned.

Collaboration usability for a stored procedure ResultSet

The Column data of the ResultSets can be dragged-and-dropped from their XSC nodes to the Business Rules. Below is a code snippet that can be generated by the Collaboration Editor:

```
while (getSPIn().getSpS_multi().resultsAvailable())
{
if (getSPIn().getSpS_multi().getUpdateCount() > 0)
  {
    System.err.println("Updated "+getSPIn().getSpS_multi().getUpdateCount()+" rows");
  }

  if (getSPIn().getSpS_multi().getNormRS().available())
  {
    while (getSPIn().getSpS_multi().getNormRS().next())
    {
      System.err.println("Customer Id   = "+getSPIn().
getSpS_multi().getNormRS().getCustomerId());
      System.err.println("Customer Name = "+getSPIn().
getSpS_multi().getNormRS().getCustomerName());
      System.err.println();
    }
    System.err.println("===");
  }
  else if (getSPIn().getSpS_multi().getDbEmployee().available())
  {
    while (getSPIn().getSpS_multi().getDbEmployee().next())
    {
      System.err.println("EMPNO    = "+getSPIn().
getSpS_multi().getDbEmployee().getEMPNO());
      System.err.println("ENAME    = "+getSPIn().
getSpS_multi().getDbEmployee().getENAME());
      System.err.println("JOB      = "+getSPIn().
getSpS_multi().getDbEmployee().getJOB());
      System.err.println("MGR      = "+getSPIn().
getSpS_multi().getDbEmployee().getMGR());
      System.err.println("HIREDATE = "+getSPIn().
getSpS_multi().getDbEmployee().getHIREDATE());
      System.err.println("SAL      = "+getSPIn().
getSpS_multi().getDbEmployee().getSAL());
      System.err.println("COMM     = "+getSPIn().
getSpS_multi().getDbEmployee().getCOMM());
      System.err.println("DEPTNO   = "+getSPIn().
getSpS_multi().getDbEmployee().getDEPTNO());
      System.err.println();
    }
    System.err.println("===");
  }
}
```

Note – resultsAvailable() and **available()** cannot be indiscriminately called because each time they move ResultSet pointers to the appropriate locations.

After calling "**resultsAvailable()**", the next result (if available) can be either a **ResultSet** or an **UpdateCount** if the default "**enableResultSetsAndUpdateCount()**" was used.

Because of limitations imposed by some DBMSs, it is recommended that for maximum portability, all of the results in a ResultSet object should be retrieved before OUT parameters are retrieved. Therefore, you should retrieve all ResultSet(s) and Update Counts first followed by retrieving the OUT type parameters and return values.

The following list includes specific ResultSet behavior that you may encounter:

- The method **resultsAvailable()** implicitly calls **getMoreResults()** when it is called more than once. You should not call both methods in your Java code. Doing so may result in skipped data from one of the ResultSets when more than one ResultSet is present.
- The methods **available()** and **getResultSet()** can not be used in conjunction with multiple ResultSets being open at the same time. Attempting to open more the one ResultSet at the same time closes the previous ResultSet. The recommended working pattern is:

  - Open one Result Set (ResultSet_1) and work with the data until you have completed your modifications and updates. Open ResultSet_2, (ResultSet_1 is now closed) and modify. When you have completed your work in ResultSet_2, open any additional ResultSets or close ResultSet_2.

  If you modify the ResultSet generated by the Execute mode of the Database Wizard, you need to assure the indexes match the stored procedure. By doing this, your ResultSet indexes are preserved.

- Generally, getMoreResults does not need to be called. It is needed if you do not want to use our enhanced methods and you want to follow the traditional JDBC calls on your own.

The DBWizard Assistant expects the column names to be in English when creating a ResultSet.

Prepared Statement

A Prepared Statement OTD represents a SQL statement that has been compiled. Fields in the OTD correspond to the input values that users need to provide.

Prepared statements can be used to perform insert, update, delete and query operations. A prepared statement uses a question mark (?) as a place holder for input. For example:

```
insert into EMP_TAB(Age, Name, Dept No) value(?, ?, ?)
```

To execute a prepared statement, set the input parameters and call **executeUpdate()** and specify the input values if any.

```
getPrepStatement().getPreparedStatementTest().setAge(23);
getPrepStatement().getPreparedStatementTest().setName(”Peter Pan’);
getPrepStatement().getPreparedStatementTest().setDeptNo(6);
getPrepStatement().getPreparedStatementTest().executeUpdate();
```

Batch Operations

To achieve better performance, consider using a bulk insert if you have to insert many records. This is the “Add Batch” capability. The only modification required is to include the **addBatch()** method for each SQL operation and then the **executeBatch()** call to submit the batch to the database server. Batch operations apply only to Prepared Statements.

```
getPrepStatement().getPreparedStatementTest().setAge(23);
getPrepStatement().getPreparedStatementTest().setName(”Peter Pan’);
getPrepStatement().getPreparedStatementTest().setDeptNo(6);
getPrepStatement().getPreparedStatementTest().addBatch();

getPrepStatement().getPreparedStatementTest().setAge(45);
getPrepStatement().getPreparedStatementTest().setName(”Harrison Ford’);
getPrepStatement().getPreparedStatementTest().setDeptNo(7);
getPrepStatement().getPreparedStatementTest().addBatch();
getPrepStatement().getPreparedStatementTest().executeBatch();
```

Using VSAM Operations

The database operations used in the VSAM Adapter are used to access the VSAM database. Database operations are either accessed through Activities in BPEL, or through methods called from a JCD Collaboration.

- "VSAM Adapter Database Operations (BPEL)" on page 30
- "VSAM Adapter Database Operations (JCD)" on page 32

VSAM Adapter Database Operations (BPEL)

The VSAM Adapter uses a number operations to query the VSAM database. Within a BPEL business process, the VSAM Adapter uses BPEL Activities to perform basic outbound database operations, including:

- Insert
- Update
- Delete
- SelectOne
- SelectMultiple
- SelectAll

In addition to these outbound operations, the VSAM Adapter also employs the inbound Activity **ReceiveOne** within a Prepared Statement OTD.

Activity Input and Output

The Sun SeeBeyond Enterprise Designer– Business Rules Designer includes Input and Output columns to map and transform data between Activities displayed on the Business Process Canvas.

Figure 1–3 displays the business rules between the **FileClient.write** and **otdVSAM.CICSEMPDelete** Activities. In this example, the **whereClause** appears on the Input side.

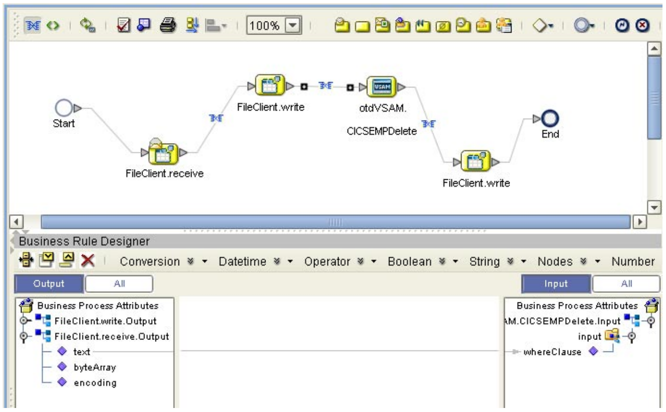


FIGURE 1–3 Input and Output Between Activities

The following table lists the expected Input and Output of each database operation Activity.

TABLE 1–3 VSAM Operations

eInsight Operations	Activity Input	Activity Output
SelectAll	where() clause (optional)	Returns all rows that fit the condition of the where() clause.
SelectMultiple	number of rows where() clause (optional)	Returns the number of rows specified that fit the condition of the where() clause, and the number of rows to be returned. For example: If the number of rows that meet the condition are 5 and the number of available rows are 10, then only 5 rows will be returned. Alternately, if the number of rows that meet the condition are 20, but if the number of available rows are 10, then only 10 rows are returned.
SelectOne	where() clause (optional)	Returns the first row that fits the condition of the where() clause.
Insert	definition of new item to be inserted	Returns status.
Update	where() clause	Returns status.
Delete	where() clause	Returns status.

VSAM Adapter Database Operations (JCD)

The same database operations are also used in the JCD, but appear as methods to call from the Collaboration.

Tables and Views are manipulated through OTDs. Methods to call include:

- insert()
- insertRow()
- update(*String sWhere*)
- updateRow()
- delete(*String sWhere*)
- deleteRow()
- select(*String where*)

Note – Refer to the Javadoc for a full description of methods included in the VSAM Adapter.

The Table

A table OTD represents a database table. It consists of fields and methods. Fields correspond to the columns of a table while methods are the operations that you can apply to the OTD. This allows you to perform query, update, insert, and delete SQL operations in a table. The ability to update via a resultset is called “Updatable Resultset”, which is a feature supported by this adapter.

By default, the Table OTD has UpdatableConcurrency and ScrollTypeForwardOnly. Normally you do not have to change the default setting.

The type of result returned by the select() method can be specified using:

- SetConcurrencytoUpdatable
- SetConcurrencytoReadOnly
- SetScrollTypetoForwardOnly
- SetScrollTypetoScrollSensitive
- SetScrollTypetoInsensitive

The Query (Select) Operation

▼ To perform a query operation on a table:

1 **Execute the select() method with the "where" clause specified if necessary.**

2 **Loop through the ResultSet using the next() method.**

3 **Process the return record within a while() loop.**

For example:

```
package prjVSAM_JCDjcdALL;

public class jcdTableSelect
{

    public com.stc.codegen.logger.Logger logger;

    public com.stc.codegen.alerter.Alerter alerter;

    public com.stc.codegen.util.CollaborationContext collabContext;

    public com.stc.codegen.util.TypeConverter typeConverter;

    public void receive( com.stc.connector.appconn.file.
FileTextMessage input, otdVSAM.OtdVSAMOTD otdVSAM_1, dtd.
otdOutputDTD_1935483687.Emp otdOutputDTD_Emp_1, com.stc.connector.
appconn.file.FileApplication FileClient_1 )
        throws Throwable
    {
        FileClient_1.setText( "Selecting record(s) from CICSEMP
table via table select .." );
        FileClient_1.write();
        otdVSAM_1.getCICSEMP().select( input.getText() );
        while (otdVSAM_1.getCICSEMP().next()) {
            otdOutputDTD_Emp_1.setENAME( otdVSAM_1.getCICSEMP().getENAME() );
            otdOutputDTD_Emp_1.setPHONE( typeConverter.intToString( otdVSAM_1.
getCICSEMP().getPHONE(), "#", false, "" ) );
            otdOutputDTD_Emp_1.setMAILID( otdVSAM_1.getCICSEMP().getMAILID() );
            otdOutputDTD_Emp_1.setSALARY( otdVSAM_1.getCICSEMP().getSALARY().
toString() );
            otdOutputDTD_Emp_1.setJOBID( typeConverter.doubleToString(
 otdVSAM_1.getCICSEMP().getJOBID(), "#.000000;-#.000000", false, "" ) );
            otdOutputDTD_Emp_1.setEMPID( typeConverter.intToString(
 otdVSAM_1.getCICSEMP().getEMPID(), "#", false, "" ) );
            otdOutputDTD_Emp_1.setDEPTID( typeConverter.shortToString(
 otdVSAM_1.getCICSEMP().getDEPTID(), "#", false, "" ) );
```

```
            otdOutputDTD_Emp_1.setDEPARTMENT( otdVSAM_1.getCICSEMP().getDEPARTMENT() );
            FileClient_1.setText( otdOutputDTD_Emp_1.marshalToString() );
            FileClient_1.write();
        }
        FileClient_1.setText( "Done table select." );
        FileClient_1.write();
    }

}
```

The Insert Operation

To perform an insert operation on a table:

1. **Execute the insert() method. Assign a field.**
2. **Insert the row by calling insertRow()**

   This example inserts an employee record:

```
package prjVSAM_JCDjcdALL;

public class jcdInsert
{

    public com.stc.codegen.logger.Logger logger;

    public com.stc.codegen.alerter.Alerter alerter;

    public com.stc.codegen.util.CollaborationContext collabContext;

    public com.stc.codegen.util.TypeConverter typeConverter;

    public void receive( com.stc.connector.appconn.file.FileTextMessage input,
 dtd.otdInputDTD_622919076.Emp otdInputDTD_Emp_1, otdVSAM.OtdVSAMOTD otdVSAM_1,
 com.stc.connector.appconn.file.FileApplication FileClient_1 )
        throws Throwable
    {
        FileClient_1.setText( "Inserting records into CICSEMP table .." );
        FileClient_1.write();
        otdInputDTD_Emp_1.unmarshalFromString( input.getText() );
        otdVSAM_1.getCICSEMP().insert();
        for (int i1 = 0; i1 < otdInputDTD_Emp_1.countX_sequence_A(); i1 += 1) {
            otdVSAM_1.getCICSEMP().setENAME(
 otdInputDTD_Emp_1.getX_sequence_A( i1 ).getENAME() );
            otdVSAM_1.getCICSEMP().setPHONE( typeConverter.stringToInt(
 otdInputDTD_Emp_1.getX_sequence_A( i1 ).getPHONE(), "#", false, 0 ) );
            otdVSAM_1.getCICSEMP().setMAILID( otdInputDTD_Emp_1.getX_sequence_A(
 i1 ).getMAILID() );
           otdVSAM_1.getCICSEMP().setSALARY( new java.math.BigDecimal(
 otdInputDTD_Emp_1.getX_sequence_A( i1 ).getSALARY() ) );
           otdVSAM_1.getCICSEMP().setJOBID( typeConverter.stringToDouble(
 otdInputDTD_Emp_1.getX_sequence_A( i1 ).getJOBID(), "#.000000;-#.000000",
 false, 0 ) ); otdVSAM_1.getCICSEMP().setEMPID( typeConverter.stringToInt(
 otdInputDTD_Emp_1.getX_sequence_A( i1 ).getEMPID(), "#", false, 0 ) );
           otdVSAM_1.getCICSEMP().setDEPTID( typeConverter.stringToShort(
 otdInputDTD_Emp_1.getX_sequence_A( i1 ).getDEPTID(), "#", false, 0 ) );
           otdVSAM_1.getCICSEMP().setDEPARTMENT( otdInputDTD_Emp_1.getX_sequence_A(
 i1 ).getDEPARTMENT() );
           otdVSAM_1.getCICSEMP().insertRow();
       }
       FileClient_1.setText( "Done Insert." );
       FileClient_1.write();
    }
}
```

The Update Operation

To perform an update operation on a table:

1. **Execute the update() method.**
2. **Using a while loop together with next(), move to the row that you want to update.**
3. **Assign updating value(s) to the fields of the table OTD**
4. **Update the row by calling updateRow().**

```
package prjVSAM_JCDjcdALL;

public class jcdUpdate
{

    public com.stc.codegen.logger.Logger logger;

    public com.stc.codegen.alerter.Alerter alerter;

    public com.stc.codegen.util.CollaborationContext collabContext;

    public com.stc.codegen.util.TypeConverter typeConverter;

    public void receive( com.stc.connector.appconn.file.FileTextMessage
input, otdVSAM.OtdVSAMOTD otdVSAM_1, dtd.otdOutputDTD_1935483687.
Emp otdOutputDTD_Emp_1, com.stc.connector.appconn.file.FileApplication FileClient_1 )
        throws Throwable
    {
        FileClient_1.setText( "Update the Department .. " );
        FileClient_1.write();
        otdVSAM_1.getCICSEMP().update( input.getText() );
        while (otdVSAM_1.getCICSEMP().next()) {
            otdVSAM_1.getCICSEMP().setDEPARTMENT( "QAQAQA" );
            otdVSAM_1.getCICSEMP().updateRow();
        }
        FileClient_1.setText( "Done Update." );
        FileClient_1.write();
    }

}
```

The Delete Operation

▼ To perform a delete operation on a table:

- **Execute the delete() method.**

  In this example DELETE an employee.

```
package prjVSAM_JCDjcdALL;

public class jcdDelete
{

    public com.stc.codegen.logger.Logger logger;

    public com.stc.codegen.alerter.Alerter alerter;

    public com.stc.codegen.util.CollaborationContext collabContext;

    public com.stc.codegen.util.TypeConverter typeConverter;

    public void receive( com.stc.connector.appconn.file.
FileTextMessage input, otdVSAM.OtdVSAMOTD otdVSAM_1, dtd.
otdOutputDTD_1935483687.Emp otdOutputDTD_Emp_1, com.stc.connector.
appconn.file.FileApplication FileClient_1 )
        throws Throwable
    {
        FileClient_1.setText( "Delete record .." );
        FileClient_1.write();
        otdVSAM_1.getCICSEMP().delete( input.getText() );
        FileClient_1.setText( "Done delete." );
        FileClient_1.write();
    }

}
```

Prepared Statement

A Prepared Statement OTD represents a SQL statement that has been compiled. Fields in the OTD correspond to the input values that users need to provide.

Prepared statements can be used to perform insert, update, delete and query operations. A prepared statement uses a question mark (?) as a place holder for input. For example:

```
insert into EMP_TAB(Age, Name, Dept No) value(?, ?, ?)
```

To execute a prepared statement, set the input parameters and call **executeUpdate()** and specify the input values if any.

```
getPrepStatement().getPreparedStatementTest().setAge(23);
getPrepStatement().getPreparedStatementTest().setName(”Peter Pan’);
getPrepStatement().getPreparedStatementTest().setDeptNo(6);
getPrepStatement().getPreparedStatementTest().executeUpdate();
```

Batch Operations

To achieve better performance, consider using a bulk insert if you have to insert many records. This is the “Add Batch” capability. The only modification required is to include the **addBatch()** method for each SQL operation and then the **executeBatch()** call to submit the batch to the database server. Batch operations apply only to Prepared Statements.

```
getPrepStatement().getPreparedStatementTest().setAge(23);
getPrepStatement().getPreparedStatementTest().setName(”Peter Pan’);
getPrepStatement().getPreparedStatementTest().setDeptNo(6);
getPrepStatement().getPreparedStatementTest().addBatch();

getPrepStatement().getPreparedStatementTest().setAge(45);
getPrepStatement().getPreparedStatementTest().setName(”Harrison Ford’);
getPrepStatement().getPreparedStatementTest().setDeptNo(7);
getPrepStatement().getPreparedStatementTest().addBatch();
getPrepStatement().getPreparedStatementTest().executeBatch();
```

Using SQL ServerOperations

The database operations used in the SQL Server Adapter are used to access the SQL Server database. Database operations are either accessed through Activities in BPEL, or through methods called from a JCD Collaboration.

- “SQL Server Adapter Database Operations (BPEL)” on page 38
- “SQL Server Adapter Database Operations (JCD)” on page 39

SQL Server Adapter Database Operations (BPEL)

The SQL Server Adapter uses a number operations to query the SQL Server database. Within a BPEL business process, the SQL Server Adapter uses BPEL Activities to perform basic outbound database operations, including:

- Insert
- Update
- Delete
- SelectOne
- SelectMultiple
- SelectAll

In addition to these outbound operations, the SQL Server Adapter also employs the inbound Activity **ReceiveOne** within a Prepared Statement OTD.

Activity Input and Output

The Sun SeeBeyond Enterprise Designer– Business Rules Designer includes Input and Output columns to map and transform data between Activities displayed on the Business Process Canvas.

Figure 1–4 displays the business rules between the **FileClient.write** and **otdSQLServer.Db_employeeDelete** Activities. In this example, the **whereClause** appears on the Input side.

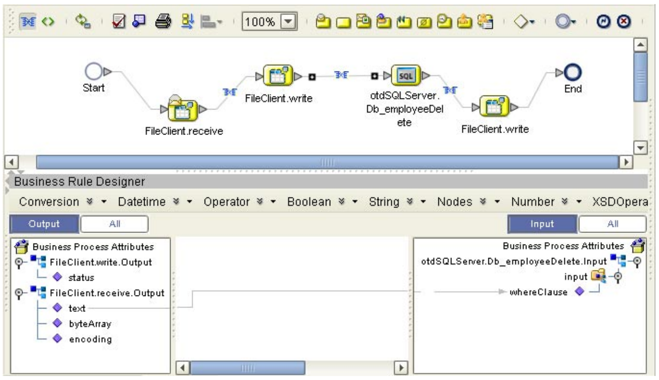


FIGURE 1–4 Input and Output Between Activities

The following table lists the expected Input and Output of each database operation Activity.

TABLE 1–4 SQL Server Operations

eInsight Operations	Activity Input	Activity Output
SelectAll	where() clause (optional)	Returns all rows that fit the condition of the where() clause.
SelectMultiple	number of rows where() clause (optional)	Returns the number of rows specified that fit the condition of the where() clause, and the number of rows to be returned. For example: If the number of rows that meet the condition are 5 and the number of available rows are 10, then only 5 rows will be returned. Alternately, if the number of rows that meet the condition are 20, but if the number of available rows are 10, then only 10 rows are returned.
SelectOne	where() clause (optional)	Returns the first row that fits the condition of the where() clause.
Insert	definition of new item to be inserted	Returns status.
Update	where() clause	Returns status.
Delete	where() clause	Returns status.

SQL Server Adapter Database Operations (JCD)

The same database operations are also used in the JCD, but appear as methods to call from the Collaboration.

Tables, Views, and Stored Procedures are manipulated through OTDs. Methods to call include:

- insert()
- insertRow()
- update(*String sWhere*)
- updateRow()
- delete(*String sWhere*)
- deleteRow()
- select(*String where*)

Note – Refer to the Javadoc for a full description of methods included in the SQL Server Adapter.

The Table

A table OTD represents a database table. It consists of fields and methods. Fields correspond to the columns of a table while methods are the operations that you can apply to the OTD. This allows you to perform query, update, insert, and delete SQL operations in a table. The ability to update via a resultset is called “Updatable Resultset”, which is a feature supported by this adapter.

By default, the Table OTD has UpdatableConcurrency and ScrollTypeForwardOnly. Normally you do not have to change the default setting.

The type of result returned by the select() method can be specified using:

- SetConcurrencytoUpdatable
- SetConcurrencytoReadOnly
- SetScrollTypetoForwardOnly
- SetScrollTypetoScrollSensitive
- SetScrollTypetoInsensitive

The Query (Select) Operation

▼ To perform a query operation on a table

1. **Execute the select() method with the “where” clause specified if necessary.**
2. **Loop through the ResultSet using the next() method.**
3. **Process the return record within a while() loop.**

   For example:

```
package prjSQLServer_JCDjcdALL;


public class jcdTableSelect
{

    public com.stc.codegen.logger.Logger logger;

    public com.stc.codegen.alerter.Alerter alerter;
    public com.stc.codegen.util.CollaborationContext collabContext;

    public com.stc.codegen.util.TypeConverter typeConverter;

    public void receive( com.stc.connector.appconn.file.
FileTextMessage input, dtd.otdOutputDTD1325973702.
DB_Employee otdOutputDTD_DB_Employee_1, otdSQLServer.OtdSQLServerOTD
otdSQLServer_1, com.stc.connector.appconn.file.FileApplication FileClient_1 )
        throws Throwable
    {
        FileClient_1.setText( "Selectiong records from db_employee table via
Table Select........" );
        FileClient_1.write();
        otdSQLServer_1.getDb_employee().select( input.getText() );
        while (otdSQLServer_1.getDb_employee().next()) {
            otdOutputDTD_DB_Employee_1.setEmpNo( typeConverter.shortToString(
otdSQLServer_1.getDb_employee().getEMP_NO(), "#", false, "" ) );
            otdOutputDTD_DB_Employee_1.setLastname(
otdSQLServer_1.getDb_employee().getLAST_NAME() );
            otdOutputDTD_DB_Employee_1.setFirstname(
otdSQLServer_1.getDb_employee().getFIRST_NAME() );
            otdOutputDTD_DB_Employee_1.setRate(
otdSQLServer_1.getDb_employee().getRATE().toString() );
            otdOutputDTD_DB_Employee_1.setLastDate(
typeConverter.dateToString( otdSQLServer_1.getDb_employee().getLAST_UPDATE(),
"yyyy-MM-dd hh:mm:ss", false, "" ) );
            FileClient_1.setText( otdOutputDTD_DB_Employee_1.marshalToString() );
            FileClient_1.write();
        }
        FileClient_1.setText( "Table Select Done." );
        FileClient_1.write();
    }

}
```

The Insert Operation

▼ To perform an insert operation on a table

1. **Execute the insert() method. Assign a field.**

2. **Insert the row by calling insertRow()**

   This example inserts an employee record.

```
package prjSQLServer_JCDjcdALL;
```

```
public class jcdInsert
{

    public com.stc.codegen.logger.Logger logger;

    public com.stc.codegen.alerter.Alerter alerter;

    public com.stc.codegen.util.CollaborationContext collabContext;

    public com.stc.codegen.util.TypeConverter typeConverter;

    public void receive( com.stc.connector.appconn.file.FileTextMessage input,
otdSQLServer.OtdSQLServerOTD otdSQLServer_1,     dtd.otdInputDTD_1206505729.
DB_Employee otdInputDTD_DB_Employee_1, com.stc.connector.appconn.file.
FileApplication FileClient_1 )
        throws Throwable
    {
        FileClient_1.setText( "Inserting records in to db_employee table......" );
        FileClient_1.write();
        otdInputDTD_DB_Employee_1.unmarshalFromString( input.getText() );
        otdSQLServer_1.getDb_employee().insert();
        for (int i1 = 0; i1 < otdInputDTD_DB_Employee_1.countX_sequence_A(); i1 += 1) {
            otdSQLServer_1.getDb_employee().setEMP_NO( typeConverter.stringToShort(
 otdInputDTD_DB_Employee_1.getX_sequence_A( i1 ).getEmpNo(), "#", false, 0 ) );
            otdSQLServer_1.getDb_employee().setLAST_NAME( otdInputDTD_DB_Employee_1.
getX_sequence_A( i1 ).getLastname() );
            otdSQLServer_1.getDb_employee().setFIRST_NAME( otdInputDTD_DB_Employee_1.
getX_sequence_A( i1 ).getFirstname() );
            otdSQLServer_1.getDb_employee().setRATE( new java.math.BigDecimal(
 otdInputDTD_DB_Employee_1.getX_sequence_A( i1 ).getRate() ) );
            otdSQLServer_1.getDb_employee().setLAST_UPDATE( typeConverter.
stringToTimestamp( otdInputDTD_DB_Employee_1.getX_sequence_A( i1 ).getLastDate(),
 "yyyy-MM-dd hh:mm:ss", false, "" ) );
            otdSQLServer_1.getDb_employee().insertRow();
        }
        FileClient_1.setText( "Insert Done." );
        FileClient_1.write();
    }

}
```

The Update Operation

▼ To perform an update operation on a table

1. **Execute the update() method.**
2. **Using a while loop together with next(), move to the row that you want to update.**
3. **Assign updating value(s) to the fields of the table OTD**
4. **Update the row by calling updateRow().**

```
package prjSQLServer_JCDjcdALL;


public class jcdUpdate
{

public com.stc.codegen.logger.Logger logger;
public com.stc.codegen.alerter.Alerter alerter;
public com.stc.codegen.util.CollaborationContext collabContext;
public com.stc.codegen.util.TypeConverter typeConverter;

    public void receive( com.stc.connector.appconn.file.FileTextMessage input,
 otdSQLServer.OtdSQLServerOTD otdSQLServer_1, com.stc.connector.appconn.file.
FileApplication FileClient_1 )
        throws Throwable
    {
        FileClient_1.setText( "Updating the Rate and Last_update fields .. " );
        FileClient_1.write();
        otdSQLServer_1.getDb_employee().update( input.getText() );
        while (otdSQLServer_1.getDb_employee().next()) {
        otdSQLServer_1.getDb_employee().setLAST_NAME( "Krishna" );
        otdSQLServer_1.getDb_employee().setFIRST_NAME( "Kishore" );
        otdSQLServer_1.getDb_employee().updateRow();
        }
        FileClient_1.setText( "Update Done." );
        FileClient_1.write();
    }

}
```

The Delete Operation

▼ To perform a delete operation on a table

- **Execute the delete() method.**

  In this example DELETE an employee.

```
package prjSQLServer_JCDjcdALL;

public class jcdDelete
{
public com.stc.codegen.logger.Logger logger;
public com.stc.codegen.alerter.Alerter alerter;
public com.stc.codegen.util.CollaborationContext collabContext;
public com.stc.codegen.util.TypeConverter typeConverter;

    public void receive( com.stc.connector.appconn.file.FileTextMessage
 input, otdSQLServer.OtdSQLServerOTD otdSQLServer_1, com.stc.connector.
appconn.file.FileApplication FileClient_1 )
        throws Throwable
    {
        FileClient_1.setText( "Deleting record..........." );
        FileClient_1.write();
        otdSQLServer_1.getDb_employee().delete( input.getText() );
        FileClient_1.setText( "Delete Done." );
        FileClient_1.write();
    }
}
```

The Stored Procedure

A Stored Procedure OTD represents a database stored procedure. Fields correspond to the arguments of a stored procedure while methods are the operations that you can apply to the OTD. It allows you to execute a stored procedure. Remember that while in the Collaboration Editor you can drag and drop nodes from the OTD into the Collaboration Editor.

Executing Stored Procedures

The OTD represents the Stored Procedure "LookUpGlobal" with two parameters, an inbound parameter (INLOCALID) and an outbound parameter (OUTGLOBALPRODUCTID). These inbound and outbound parameters are generated by the DataBase Wizard and are represented in the resulting OTD as nodes. Within the Transformation Designer, you can drag values from the input parameters, execute the call, collect data, and drag the values to the output parameters.

Below are the steps for executing the Stored Procedure:

1. Specify the input values.
2. Execute the Stored Procedure.
3. Retrieve the output parameters if any.

For example:

```
package Storedprocedure;

public class sp_jce
{

    public com.stc.codegen.logger.Logger logger;

    public com.stc.codegen.alerter.Alerter alerter;

    public void receive( com.stc.connector.appconn.file.FileTextMessage
 input,com.stc.connector.appconn.file.FileApplication FileClient_1,employeedb.
Db_employee employeedb_with_top_db_employee_1,insert_DB.Insert_DBOTD insert_DB_1 )
    throws Throwable
    {
        //@map:employeedb_with_top_db_employee_1.unmarshalFromString(Text)
        employeedb_with_top_db_employee_1.unmarshalFromString( input.getText() );

        //@map:Copy java.lang.Integer.parseInt(Employee_no) to Employee_no
        insert_DB_1.getInsert_new_employee().setEmployee_no( java.lang.Integer.
parseInt( employeedb_with_top_db_employee_1.getEmployee_no() ) );

        //@map:Copy Employee_lname to Employee_Lname
        insert_DB_1.getInsert_new_employee().setEmployee_Lname(
 employeedb_with_top_db_employee_1.getEmployee_lname() );

        //@map:Copy Employee_fname to Employee_Fname
        insert_DB_1.getInsert_new_employee().setEmployee_Fname(
 employeedb_with_top_db_employee_1.getEmployee_fname() );

        //@map:Copy java.lang.Float.parseFloat(Rate) to Rate
        insert_DB_1.getInsert_new_employee().setRate(
 java.lang.Float.parseFloat( employeedb_with_top_db_employee_1.getRate() ) );

        //@map:Copy java.sql.Timestamp.valueOf(Update_date) to Update_date
        insert_DB_1.getInsert_new_employee().setUpdate_date(
 java.sql.Timestamp.valueOf( employeedb_with_top_db_employee_1.getUpdate_date() ) );

        //@map:Insert_new_employee.execute
        insert_DB_1.getInsert_new_employee().execute();

        //@map:insert_DB_1.commit
        insert_DB_1.commit();
        //@map:Copy "procedure executed" to Text
        FileClient_1.setText( "procedure executed" );

        //@map:FileClient_1.write
        FileClient_1.write();
    }

}
```

Manipulating the ResultSet and Update Count Returned by Stored Procedure

For Stored Procedures that return ResultSets and Update Count, the following methods are provided to manipulate the ResultSet:

- enableResultSetOnly
- enableUpdateCountsOnly
- enableResultSetandUpdateCounts
- resultsAvailable
- next
- getUpdateCount
- available

SQL Server stored procedures do not return records as ResultSets, instead, the records are returned through output reference cursor parameters. Reference Cursor parameters are essentially ResultSets.

The **resultsAvailable()** method, added to the PreparedStatementAgent class, simplifies the whole process of determining whether any results, be it Update Counts or ResultSets, are available after a stored procedure has been executed. Although JDBC provides three methods (**getMoreResults()**, **getUpdateCount()**, and **getResultSet()**) to access the results of a stored procedure call, the information returned from these methods can be quite confusing to the inexperienced Java JDBC programmer and they also differ between vendors. You can simply call **resultsAvailable()** and if Boolean true is returned, you can expect either a valid Update Count when **getUpdateCount()** is called and/or the next ResultSet has been retrieved and made available to one of the ResultSet nodes defined for the Stored Procedure OTD, when that node's **available()** method returns true.

Frequently, Update Counts information that is returned from a Stored Procedures is insignificant. You should process returned ResultSet information and avoid looping through all of the Update Counts. The following three methods control exactly what information should be returned from a stored procedure call. The **enableResultSetsOnly()** method, added to the PreparedStatement Agent class allows only ResultSets to be returned and thus every **resultsAvailable()** called only returns Boolean true if a ResultSet is available. Likewise, the

enableUpdateCountsOnly() causes **resultsAvailable()** to return true only if an Update Count is available. The default case of **enableResultsetsAndUpdateCount()** method allows both ResultSets and Update Counts to be returned.

Collaboration usability for a stored procedure ResultSet

The Column data of the ResultSets can be dragged-and-dropped from their XSC nodes to the Business Rules. Below is a code snippet that can be generated by the Collaboration Editor:

```
while (getSPIn().getSpS_multi().resultsAvailable())
{
if (getSPIn().getSpS_multi().getUpdateCount() > 0)
  {
    System.err.println("Updated "+getSPIn().getSpS_multi().getUpdateCount()+" rows");
  }

  if (getSPIn().getSpS_multi().getNormRS().available())
  {
    while (getSPIn().getSpS_multi().getNormRS().next())
    {
      System.err.println("Customer Id   = "+getSPIn().getSpS_multi().getNormRS().
getCustomerId());
      System.err.println("Customer Name = "+getSPIn().getSpS_multi().getNormRS().
getCustomerName());
      System.err.println();
    }
    System.err.println("===");
  }
  else if (getSPIn().getSpS_multi().getDbEmployee().available())
  {
    while (getSPIn().getSpS_multi().getDbEmployee().next())
    {
      System.err.println("EMPNO    = "+getSPIn().getSpS_multi().
getDbEmployee().getEMPNO());
      System.err.println("ENAME    = "+getSPIn().getSpS_multi().
getDbEmployee().getENAME());
      System.err.println("JOB      = "+getSPIn().getSpS_multi().
getDbEmployee().getJOB());
      System.err.println("MGR      = "+getSPIn().getSpS_multi().
getDbEmployee().getMGR());
      System.err.println("HIREDATE = "+getSPIn().getSpS_multi().
getDbEmployee().getHIREDATE());
      System.err.println("SAL      = "+getSPIn().getSpS_multi().
getDbEmployee().getSAL());
      System.err.println("COMM     = "+getSPIn().getSpS_multi().
getDbEmployee().getCOMM());
      System.err.println("DEPTNO   = "+getSPIn().getSpS_multi().
getDbEmployee().getDEPTNO());
```

```
      System.err.println();
    }
    System.err.println("===");
  }
}
```

Note – resultsAvailable() and **available()** cannot be indiscriminately called because each time they move ResultSet pointers to the appropriate locations.

After calling "**resultsAvailable()**", the next result (if available) can be either a **ResultSet** or an **UpdateCount** if the default "**enableResultSetsAndUpdateCount()**" was used.

Because of limitations imposed by some DBMSs, it is recommended that for maximum portability, all of the results in a ResultSet object should be retrieved before OUT parameters are retrieved. Therefore, you should retrieve all ResultSet(s) and Update Counts first followed by retrieving the OUT type parameters and return values.

The following list includes specific ResultSet behavior that you may encounter:

- The method **resultsAvailable()** implicitly calls **getMoreResults()** when it is called more than once. You should not call both methods in your java code. Doing so may result in skipped data from one of the ResultSets when more than one ResultSet is present.
- The methods **available()** and **getResultSet()** can not be used in conjunction with multiple ResultSets being open at the same time. Attempting to open more the one ResultSet at the same time closes the previous ResultSet. The recommended working pattern is:
  - Open one Result Set (ResultSet_1) and work with the data until you have completed your modifications and updates. Open ResultSet_2, (ResultSet_1 is now closed) and modify. When you have completed your work in ResultSet_2, open any additional ResultSets or close ResultSet_2.

  If you modify the ResultSet generated by the Execute mode of the Database Wizard, you need to assure the indexes match the stored procedure. By doing this, your ResultSet indexes are preserved.
- Generally, getMoreResults does not need to be called. It is needed if you do not want to use our enhanced methods and you want to follow the traditional JDBC calls on your own.

The DBWizard Assistant expects the column names to be in English when creating a ResultSet.

Prepared Statement

A Prepared Statement OTD represents a SQL statement that has been compiled. Fields in the OTD correspond to the input values that users need to provide.

Prepared statements can be used to perform insert, update, delete and query operations. A prepared statement uses a question mark (?) as a place holder for input. For example:

```
insert into EMP_TAB(Age, Name, Dept No) value(?, ?, ?)
```

To execute a prepared statement, set the input parameters and call **executeUpdate()** and specify the input values if any.

```
getPrepStatement().getPreparedStatementTest().setAge(23);
getPrepStatement().getPreparedStatementTest().setName(”Peter Pan’);
getPrepStatement().getPreparedStatementTest().setDeptNo(6);
getPrepStatement().getPreparedStatementTest().executeUpdate();
```

Batch Operations

To achieve better performance, consider using a bulk insert if you have to insert many records. This is the “Add Batch” capability. The only modification required is to include the **addBatch()** method for each SQL operation and then the **executeBatch()** call to submit the batch to the database server. Batch operations apply only to Prepared Statements.

```
getPrepStatement().getPreparedStatementTest().setAge(23);
getPrepStatement().getPreparedStatementTest().setName(”Peter Pan’);
getPrepStatement().getPreparedStatementTest().setDeptNo(6);
getPrepStatement().getPreparedStatementTest().addBatch();

getPrepStatement().getPreparedStatementTest().setAge(45);
getPrepStatement().getPreparedStatementTest().setName(”Harrison Ford’);
getPrepStatement().getPreparedStatementTest().setDeptNo(7);
getPrepStatement().getPreparedStatementTest().addBatch();
getPrepStatement().getPreparedStatementTest().executeBatch();
```

Result Sets

The SQL Server Adapter supports the following 3 types of Result Sets:

- Forward Only
- Scroll Insensitive
- Scroll Sensitive

By default, the SQL Server Adapter uses Forward only. To traverse backward, you must explicitly set the result set type as scroll sensitive or scroll insensitive according to the requirement.

Type_Forward_Only	The result set is nonscrollable; its cursor moves forward only, from top to bottom. The view of the data in the result set depends on whether the DBMS materializes results incrementally.
Type_Scroll_Insensitive	The result set is scrollable: Its cursor can move forward or backward and can be moved to a particular row or to a row whose position is relative to its current position. The result set generally does not show changes to the underlying database that are made

	while it is open. The membership, order, and column values of rows are typically fixed when the result set is created.
Type_Scroll_Sensitive	The result set is scrollable; its cursor can move forward or backward and can be moved to a particular row or to a row whose position is relative to its current position. The result set is sensitive to changes made while it is open. If the underlying column values are modified, the new values are visible, thus providing a dynamic view of the underlying data. The membership and ordering of rows in the result set may be fixed or not, depending on the implementation.

Using Result Sets with stored procedures:

In order to scroll back in a result set returned from a stored procedure, the <OTD>.setScrollTypeToScrollInsensitive() method should be called before any other methods for the OTD.

Example:

Sch_StoredProcedures.setScrollTypeToScrollInsensitive();

Sch_StoredProcedures.getUsp_GetAppointment().setPatientID(PatientID);

Sch_StoredProcedures.getUsp_GetAppointment().setAppointmentDate(ApointmentDateTime);

Sch_StoredProcedures.getUsp_GetAppointment().execute();

Sch_StoredProcedures.getUsp_GetAppointment().enableResultSetsOnly();

If the <OTD>.setScrollTypeToScrollInsensitive() method is not called, then any attempt to move back in the resultset will fail.

Example:

Executing 'Sch_StoredProcedures.getUsp_GetAppointment().get$usp_GetAppointmentResultSets0().first();' will fail with the exception: 'Unsupported method: ResultSet.first'.

Using DB2 Operations

The database operations used in the DB2 are used to access the DB2 database. Database operations are either accessed through Activities in BPEL, or through methods called from a JCD Collaboration.

- “DB2 Database Operations (BPEL)” on page 51
- “DB2 Database Operations (JCD)” on page 53

DB2 Database Operations (BPEL)

The DB2 uses a number operations to query the DB2 database. Within a BPEL business process, the DB2 uses BPEL Activities to perform basic outbound database operations, including:

- Insert
- Update
- Delete
- SelectOne
- SelectMultiple
- SelectAll

In addition to these outbound operations, the DB2 Adapter also employs the inbound Activity **ReceiveOne** within a Prepared Statement OTD.

Activity Input and Output

The Sun SeeBeyond Enterprise Designer– Business Rules Designer includes Input and Output columns to map and transform data between Activities displayed on the Business Process Canvas.

Figure 1–5 displays the business rules between the **FileClient.write** and **otdDB2.Db_employeeDelete** Activities. In this example, the **whereClause** appears on the Input side.

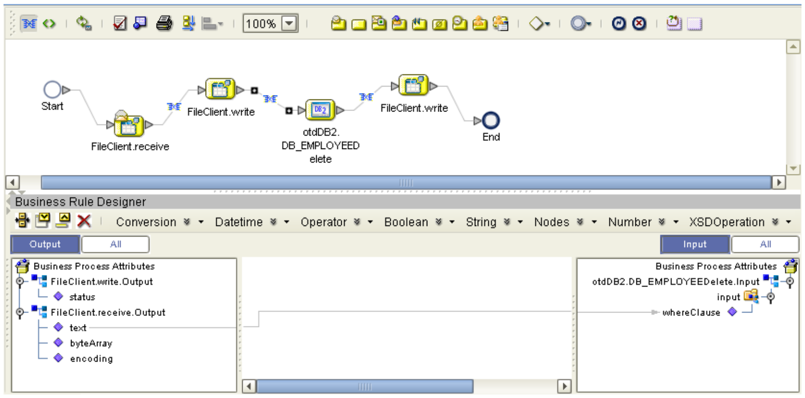


FIGURE 1–5 Input and Output Between Activities

The following table lists the expected Input and Output of each database operation Activity.

TABLE 1–5 DB2 Operations

eInsight Operations	Activity Input	Activity Output
SelectAll	where() clause (optional)	Returns all rows that fit the condition of the where() clause.
SelectMultiple	number of rows where() clause (optional)	Returns the number of rows specified that fit the condition of the where() clause, and the number of rows to be returned. For example: If the number of rows that meet the condition are 5 and the number of available rows are 10, then only 5 rows will be returned. Alternately, if the number of rows that meet the condition are 20, but if the number of available rows are 10, then only 10 rows are returned.
SelectOne	where() clause (optional)	Returns the first row that fits the condition of the where() clause.
Insert	definition of new item to be inserted	Returns status.
Update	where() clause	Returns status.
Delete	where() clause	Returns status.

DB2 Database Operations (JCD)

The same database operations are also used in the JCD, but appear as methods to call from the Collaboration.

Tables, Views, and Stored Procedures are manipulated through OTDs. Methods to call include:

- insert()
- insertRow()
- update(*String sWhere*)
- updateRow()
- delete(*String sWhere*)
- deleteRow()
- select(*String where*)

Note – Refer to the Javadoc for a full description of methods included in the [Please define the ProductName text entity].

The Table

A table OTD represents a database table. It consists of fields and methods. Fields correspond to the columns of a table while methods are the operations that you can apply to the OTD. This allows you to perform query, update, insert, and delete SQL operations in a table. The ability to update via a resultset is called "Updatable Resultset", which is a feature supported by this adapter.

By default, the Table OTD has UpdatableConcurrency and ScrollTypeForwardOnly. Normally you do not have to change the default setting.

The type of result returned by the select() method can be specified using:

- SetConcurrencytoUpdatable
- SetConcurrencytoReadOnly
- SetScrollTypetoForwardOnly
- SetScrollTypetoScrollSensitive
- SetScrollTypetoInsensitive

The Query (Select) Operation

▼ To Perform a Query Operation on a Table

1 **Execute the select() method with the "where" clause specified if necessary.**

Note – The content of the input.getText() file may contain null, meaning it will not have a "where" clause or it can contain a "where" clause such as empno > 50.

2 **Loop through the ResultSet using the next() method.**

3 **Process the return record within a while() loop.**

For example:

```
package prjDB2_JCDjcdALL;


public class jcdTableSelect
{

    public com.stc.codegen.logger.Logger logger;

    public com.stc.codegen.alerter.Alerter alerter;

    public com.stc.codegen.util.CollaborationContext collabContext;

    public com.stc.codegen.util.TypeConverter typeConverter;

    public void receive( com.stc.connector.appconn.file.FileTextMessage input,
 dtd.otdOutputDTD1325973702.
B_Employee otdOutputDTD_DB_Employee_1, otdDB2.OtdDB2OTD otdDB2_1,
com.stc.connector.appconn.
file.FileApplication FileClient_1 )
        throws Throwable
    {
        FileClient_1.setText( "Selectiong records from db_employee table via
Table Select......." );
        FileClient_1.write();
        otdDB2_1.getDb_employee().select( input.getText() );
        while (otdDB2_1.getDb_employee().next()) {
            otdOutputDTD_DB_Employee_1.setEmpNo( typeConverter.shortToString
( otdDB2_1.getDb_employee().getEMP_NO(),
 "#", false, "" ) );
            otdOutputDTD_DB_Employee_1.setLastname( otdDB2_1.getDb_employee().
getLAST_NAME() );
            otdOutputDTD_DB_Employee_1.setFirstname( otdDB2_1.getDb_employee().
getFIRST_NAME() );
            otdOutputDTD_DB_Employee_1.setRate( otdDB2_1.getDb_employee().
getRATE().toString() );
            otdOutputDTD_DB_Employee_1.setLastDate( typeConverter.dateToString
( otdDB2_1.getDb_employee().
getLAST_UPDATE(), "yyyy-MM-dd hh:mm:ss", false, "" ) );
            FileClient_1.setText( otdOutputDTD_DB_Employee_1.marshalToString() );
            FileClient_1.write();
        }
        FileClient_1.setText( "Table Select Done." );
        FileClient_1.write();
    }

}
```

The Insert Operation

▼ To Perform an Insert Operation on a Table

1 Execute the insert() method. Assign a field.

2 Insert the row by calling insertRow()

This example inserts an employee record.

```
package prjDB2_JCDjcdALL;


public class jcdInsert
{

    public com.stc.codegen.logger.Logger logger;

    public com.stc.codegen.alerter.Alerter alerter;

    public com.stc.codegen.util.CollaborationContext collabContext;

    public com.stc.codegen.util.TypeConverter typeConverter;

    public void receive( com.stc.connector.appconn.file.FileTextMessage
input, otdDB2.OtdDB2OTD otdDB2_1,     dtd.otdInputDTD_1206505729.
DB_Employee otdInputDTD_DB_Employee_1, com.stc.connector.appconn.file.
FileApplication FileClient_1 )
        throws Throwable
    {
        FileClient_1.setText( "Inserting records in to db_employee table......" );
        FileClient_1.write();
        otdInputDTD_DB_Employee_1.unmarshalFromString( input.getText() );
        otdDB2_1.getDb_employee().insert();
        for (int i1 = 0; i1 < otdInputDTD_DB_Employee_1.countX_sequence_A(); i1 += 1) {
            otdDB2_1.getDb_employee().setEMP_NO( typeConverter.stringToShort
( otdInputDTD_DB_Employee_1.getX_sequence_A( i1 ).getEmpNo(), "#", false, 0 ) );
            otdDB2_1.getDb_employee().setLAST_NAME( otdInputDTD_DB_Employee_1.
getX_sequence_A( i1 ).getLastname() );
            otdDB2_1.getDb_employee().setFIRST_NAME( otdInputDTD_DB_Employee_1.
getX_sequence_A( i1 ).getFirstname() );
            otdDB2_1.getDb_employee().setRATE( new java.math.BigDecimal
( otdInputDTD_DB_Employee_1.getX_sequence_A( i1 ).getRate() ) );
            otdDB2_1.getDb_employee().setLAST_UPDATE( typeConverter.stringToTimestamp
( otdInputDTD_DB_Employee_1.getX_sequence_A( i1 ).getLastDate(), "yyyy-MM-dd hh:mm:ss",
 false, "" ) );
            otdDB2_1.getDb_employee().insertRow();
        }
        FileClient_1.setText( "Insert Done." );
        FileClient_1.write();
    }

}
```

The Update Operation

▼ To Perform an Update Operation on a Table

1. **Execute the update() method.**

Note – The content of the input.getText() file may contain null, meaning it will not have a "where" clause or it can contain a "where" clause such as empno > 50.

---

2. **Using a while loop together with next(), move to the row that you want to update.**

3. **Assign updating value(s) to the fields of the table OTD**

4. **Update the row by calling updateRow().**

```
package prjDB2_JCDjcdALL;


public class jcdUpdate
{

public com.stc.codegen.logger.Logger logger;
public com.stc.codegen.alerter.Alerter alerter;
public com.stc.codegen.util.CollaborationContext collabContext;
public com.stc.codegen.util.TypeConverter typeConverter;

    public void receive( com.stc.connector.appconn.file.FileTextMessage input,
otdDB2.OtdDB2OTD otdDB2_1, com.stc.connector.appconn.file.FileApplication
FileClient_1 )
        throws Throwable
    {
        FileClient_1.setText( "Updating the Rate and Last_update fields .. " );
        FileClient_1.write();
        otdDB2_1.getDb_employee().update( input.getText() );
        while (otdDB2_1.getDb_employee().next()) {
        otdDB2_1.getDb_employee().setLAST_NAME( "Krishna" );
        otdDB2_1.getDb_employee().setFIRST_NAME( "Kishore" );
        otdDB2_1.getDb_employee().updateRow();
        }
        FileClient_1.setText( "Update Done." );
        FileClient_1.write();
    }

}
```

The Delete Operation

▼ To Perform a Delete Operation on a Table

● **Execute the delete() method.**

Note – The content of the input.getText() file may contain null, meaning it will not have a “where” clause or it can contain a “where” clause such as empno > 50.

In this example DELETE an employee.

```
package prjDB2_JCDjcdALL;

public class jcdDelete
{
public com.stc.codegen.logger.Logger logger;
public com.stc.codegen.alerter.Alerter alerter;
public com.stc.codegen.util.CollaborationContext collabContext;
public com.stc.codegen.util.TypeConverter typeConverter;

    public void receive( com.stc.connector.appconn.file.FileTextMessage input,
otdDB2.OtdDB2OTD otdDB2_1, com.stc.connector.appconn.file.FileApplication
FileClient_1 )
        throws Throwable
    {
        FileClient_1.setText( "Deleting record..........." );
        FileClient_1.write();
        otdDB2_1.getDb_employee().delete( input.getText() );
        FileClient_1.setText( "Delete Done." );
        FileClient_1.write();
    }
}
```

The Stored Procedure

A Stored Procedure OTD represents a database stored procedure. Fields correspond to the arguments of a stored procedure while methods are the operations that you can apply to the OTD. It allows you to execute a stored procedure. Remember that while in the Collaboration Editor you can drag and drop nodes from the OTD into the Collaboration Editor.

Executing Stored Procedures

The OTD represents the Stored Procedure “LookUpGlobal” with two parameters, an inbound parameter (INLOCALID) and an outbound parameter (OUTGLOBALPRODUCTID). These inbound and outbound parameters are generated by the DataBase Wizard and are represented in the resulting OTD as nodes. Within the Transformation Designer, you can drag values from the input parameters, execute the call, collect data, and drag the values to the output parameters.

1. Specify the input values.
2. Execute the Stored Procedure.
3. Retrieve the output parameters if any.

For example:

```
package Storedprocedure;

public class sp_jce
{

    public com.stc.codegen.logger.Logger logger;

    public com.stc.codegen.alerter.Alerter alerter;

    public void receive( com.stc.connector.appconn.file.FileTextMessage input,
com.stc.connector.appconn.file.FileApplication FileClient_1,employeedb.
Db_employee employeedb_with_top_db_employee_1,insert_DB.Insert_DBOTD insert_DB_1 )
    throws Throwable
    {

        employeedb_with_top_db_employee_1.unmarshalFromString( input.getText() );

        insert_DB_1.getInsert_new_employee().setEmployee_no( java.lang.Integer.
parseInt( employeedb_with_top_db_employee_1.getEmployee_no() ) );

        insert_DB_1.getInsert_new_employee().setEmployee_Lname
( employeedb_with_top_db_employee_1.getEmployee_lname() );

        insert_DB_1.getInsert_new_employee().setEmployee_Fname
( employeedb_with_top_db_employee_1.getEmployee_fname() );
```

```
        insert_DB_1.getInsert_new_employee().setRate( java.lang.Float.parseFloat
( employeedb_with_top_db_employee_1.getRate() ) );

        insert_DB_1.getInsert_new_employee().setUpdate_date( java.sql.Timestamp.valueOf
( employeedb_with_top_db_employee_1.getUpdate_date() ) );

        insert_DB_1.getInsert_new_employee().execute();

        insert_DB_1.commit();

        FileClient_1.setText( "procedure executed" );

        FileClient_1.write();
    }

}
```

Manipulating the ResultSet and Update Count Returned by Stored Procedure

For Stored Procedures that return ResultSets and Update Count, the following methods are provided to manipulate the ResultSet:

- enableResultSetOnly
- enableUpdateCountsOnly
- enableResultSetandUpdateCounts
- resultsAvailable
- next
- getUpdateCount
- available

DB2 stored procedures do not return records as ResultSets, instead, the records are returned through output reference cursor parameters. Reference Cursor parameters are essentially ResultSets.

The **resultsAvailable()** method, added to the PreparedStatementAgent class, simplifies the whole process of determining whether any results, be it Update Counts or ResultSets, are available after a stored procedure has been executed. Although JDBC provides three methods (**getMoreResults()**, **getUpdateCount()**, and **getResultSet()**) to access the results of a stored procedure call, the information returned from these methods can be quite confusing to the inexperienced Java JDBC programmer and they also differ between vendors. You can simply call **resultsAvailable()** and if Boolean true is returned, you can expect either a valid Update Count when **getUpdateCount()** is called and/or the next ResultSet has been retrieved and made available to one of the ResultSet nodes defined for the Stored Procedure OTD, when that node's **available()** method returns true.

Frequently, Update Counts information that is returned from a Stored Procedures is insignificant. You should process returned ResultSet information and avoid looping through all of the Update Counts. The following three methods control exactly what information should be returned from a stored procedure call. The **enableResultSetsOnly()** method, added to the PreparedStatement Agent class allows only ResultSets to be returned and thus every **resultsAvailable()** called only returns Boolean true if a ResultSet is available. Likewise, the **enableUpdateCountsOnly()** causes **resultsAvailable()** to return true only if an Update Count is available. The default case of **enableResultsetsAndUpdateCount()** method allows both ResultSets and Update Counts to be returned.

Collaboration usability for a stored procedure ResultSet

The Column data of the ResultSets can be dragged-and-dropped from their nodes to the Business Rules. Below is a code snippet that can be generated by the Collaboration Editor:

```
while (getSPIn().getSpS_multi().resultsAvailable())
{
if (getSPIn().getSpS_multi().getUpdateCount() > 0)
  {
    System.err.println("Updated "+getSPIn().getSpS_multi()
.getUpdateCount()+" rows");
  }

  if (getSPIn().getSpS_multi().getNormRS().available())
  {
    while (getSPIn().getSpS_multi().getNormRS().next())
    {
      System.err.println("Customer Id   = "+getSPIn().getSpS_multi().
getNormRS().getCustomerId());
      System.err.println("Customer Name = "+getSPIn().getSpS_multi()
getNormRS().getCustomerName());
      System.err.println();
    }
    System.err.println("===");
  }
  else if (getSPIn().getSpS_multi().getDbEmployee().available())
  {
    while (getSPIn().getSpS_multi().getDbEmployee().next())
    {
      System.err.println("EMPNO    = "+getSPIn().getSpS_multi().
getDbEmployee().getEMPNO());
      System.err.println("ENAME    = "+getSPIn().getSpS_multi().
getDbEmployee().getENAME());
      System.err.println("JOB      = "+getSPIn().getSpS_multi().
getDbEmployee().getJOB());
      System.err.println("MGR      = "+getSPIn().getSpS_multi().
getDbEmployee().getMGR());
```

```
      System.err.println("HIREDATE = "+getSPIn().getSpS_multi().
getDbEmployee().getHIREDATE());
      System.err.println("SAL      = "+getSPIn().getSpS_multi().
getDbEmployee().getSAL());
      System.err.println("COMM     = "+getSPIn().getSpS_multi().
getDbEmployee().getCOMM());
      System.err.println("DEPTNO   = "+getSPIn().getSpS_multi().
getDbEmployee().getDEPTNO());
      System.err.println();
    }
    System.err.println("===");
  }
}
```

Note – resultsAvailable() and **available()** cannot be indiscriminately called because each time they move ResultSet pointers to the appropriate locations.

After calling "**resultsAvailable()**", the next result (if available) can be either a **ResultSet** or an **UpdateCount** if the default "**enableResultSetsAndUpdateCount()**" was used.

Because of limitations imposed by some DBMSs, it is recommended that for maximum portability, all of the results in a ResultSet object should be retrieved before OUT parameters are retrieved. Therefore, you should retrieve all ResultSet(s) and Update Counts first followed by retrieving the OUT type parameters and return values.

The following list includes specific ResultSet behavior that you may encounter:

- The method **resultsAvailable()** implicitly calls **getMoreResults()** when it is called more than once. You should not call both methods in your java code. Doing so may result in skipped data from one of the ResultSets when more than one ResultSet is present.
- The methods **available()** and **getResultSet()** can not be used in conjunction with multiple ResultSets being open at the same time. Attempting to open more the one ResultSet at the same time closes the previous ResultSet. The recommended working pattern is:
  - Open one Result Set (ResultSet_1) and work with the data until you have completed your modifications and updates. Open ResultSet_2, (ResultSet_1 is now closed) and modify. When you have completed your work in ResultSet_2, open any additional ResultSets or close ResultSet_2.

  If you modify the ResultSet generated by the Execute mode of the Database Wizard, you need to assure the indexes match the stored procedure. By doing this, your ResultSet indexes are preserved.
- Generally, getMoreResults does not need to be called. It is needed if you do not want to use our enhanced methods and you want to follow the traditional JDBC calls on your own.

The DBWizard Assistant expects the column names to be in English when creating a ResultSet.

Prepared Statement

A Prepared Statement OTD represents a SQL statement that has been compiled. Fields in the OTD correspond to the input values that users need to provide.

Prepared statements can be used to perform insert, update, delete and query operations. A prepared statement uses a question mark (?) as a place holder for input. For example:

```
insert into EMP_TAB(Age, Name, Dept No) value(?, ?, ?)
```

To execute a prepared statement, set the input parameters and call **executeUpdate()** and specify the input values if any.

```
getPrepStatement().getPreparedStatementTest().setAge(23);
getPrepStatement().getPreparedStatementTest().setName(”Peter Pan’);
getPrepStatement().getPreparedStatementTest().setDeptNo(6);
getPrepStatement().getPreparedStatementTest().executeUpdate();
```

Batch Operations

To achieve better performance, consider using a bulk insert if you have to insert many records. This is the “Add Batch” capability. The only modification required is to include the **addBatch()** method for each SQL operation and then the **executeBatch()** call to submit the batch to the database server. Batch operations apply only to Prepared Statements.

```
getPrepStatement().getPreparedStatementTest().setAge(23);
getPrepStatement().getPreparedStatementTest().setName(”Peter Pan’);
getPrepStatement().getPreparedStatementTest().setDeptNo(6);
getPrepStatement().getPreparedStatementTest().addBatch();

getPrepStatement().getPreparedStatementTest().setAge(45);
getPrepStatement().getPreparedStatementTest().setName(”Harrison Ford’);
getPrepStatement().getPreparedStatementTest().setDeptNo(7);
getPrepStatement().getPreparedStatementTest().addBatch();
getPrepStatement().getPreparedStatementTest().executeBatch();
```

Using DB2 Connect Operations

The database operations used in the DB2 Connect Adapter are used to access the DB2 Connect database. Database operations are either accessed through Activities in BPEL, or through methods called from a JCD Collaboration.

- “DB2 Connect Adapter Database Operations (BPEL)” on page 63
- “DB2 Connect Adapter Database Operations (JCD)” on page 65

DB2 Connect Adapter Database Operations (BPEL)

The DB2 Connect Adapter uses a number operations to query the DB2 Connect database. Within a BPEL business process, the DB2 Connect Adapter uses BPEL Activities to perform basic outbound database operations, including:

- Insert
- Update
- Delete
- SelectOne
- SelectMultiple
- SelectAll

In addition to these outbound operations, the DB2 Connect Adapter also employs the inbound Activity **ReceiveOne** within a Prepared Statement OTD.

Activity Input and Output

The Sun SeeBeyond Enterprise Designer– Business Rules Designer includes Input and Output columns to map and transform data between Activities displayed on the Business Process Canvas.

Figure 1–6 displays the business rules between the **FileClient.write** and **otdDB2Connect.Db_employeeDelete** Activities. In this example, the **whereClause** appears on the Input side.

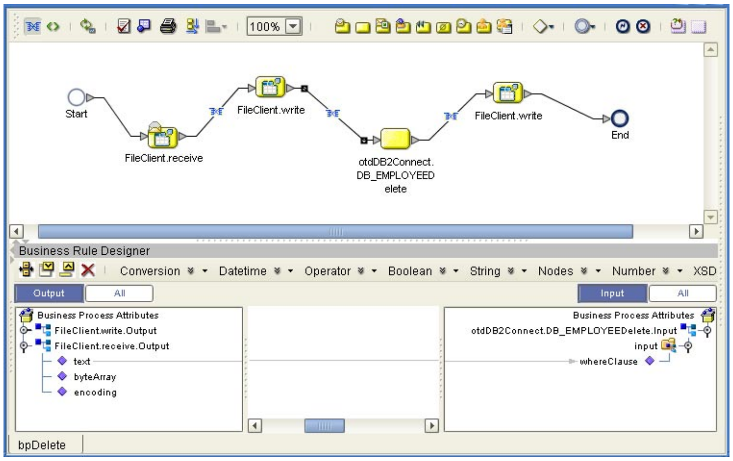


FIGURE 1–6 Input and Output Between Activities

The following table lists the expected Input and Output of each database operation Activity.

TABLE 1–6 DB2 Connect Operations

eInsight Operations	Activity Input	Activity Output
SelectAll	where() clause (optional)	Returns all rows that fit the condition of the where() clause.
SelectMultiple	number of rows where() clause (optional)	Returns the number of rows specified that fit the condition of the where() clause, and the number of rows to be returned. For example: If the number of rows that meet the condition are 5 and the number of available rows are 10, then only 5 rows will be returned. Alternately, if the number of rows that meet the condition are 20, but if the number of available rows are 10, then only 10 rows are returned.
SelectOne	where() clause (optional)	Returns the first row that fits the condition of the where() clause.
Insert	definition of new item to be inserted	Returns status.
Update	where() clause	Returns status.

TABLE 1–6 DB2 Connect Operations *(Continued)*

eInsight Operations	Activity Input	Activity Output
Delete	where() clause	Returns status.

DB2 Connect Adapter Database Operations (JCD)

The same database operations are also used in the JCD, but appear as methods to call from the Collaboration.

Tables, Views, and Stored Procedures are manipulated through OTDs. Methods to call include:

- update(*String sWhere*)
- updateRow()
- delete(*String sWhere*)
- deleteRow()
- select(*String where*)

Note – Refer to the Javadoc for a full description of methods included in the DB2 Connect Adapter.

The Table

A table OTD represents a database table. It consists of fields and methods. Fields correspond to the columns of a table while methods are the operations that you can apply to the OTD. This allows you to perform query, update, insert, and delete SQL operations in a table. The ability to update via a resultset is called "Updatable Resultset", which is a feature supported by this adapter if the Type 4 Universal driver is used (for alternate methods for the Type 2 Legacy driver refer to "Prepared Statement" on page 69).

Note – The DB2 Connect Universal Driver only supports Updatable Resultsets for Update and Delete. The Insert operation is not supported. You can use a Prepared Statement to perform the Insert operation.

By default, the Table OTD has UpdatableConcurrency and ScrollTypeForwardOnly. Normally you do not have to change the default setting.

The type of result returned by the select() method can be specified using:

- SetConcurrencytoUpdatable
- SetConcurrencytoReadOnly

- SetScrollTypetoForwardOnly
- SetScrollTypetoScrollSensitive
- SetScrollTypetoInsensitive

The Query (Select) Operation

▼ To perform a query operation on a table

1 **Execute the select() method with the "where" clause specified if necessary.**

Note – The content of the input.getText() file may contain null, meaning it will not have a "where" clause or it can contain a "where" clause such as empno > 50.

---

2 **Loop through the ResultSet using the next() method.**

3 **Process the return record within a while() loop.**

For example:

```
package prjDB2Connect_JCDjcdALL;


public class jcdTableSelect
{

    public com.stc.codegen.logger.Logger logger;

    public com.stc.codegen.alerter.Alerter alerter;

    public com.stc.codegen.util.CollaborationContext collabContext;

    public com.stc.codegen.util.TypeConverter typeConverter;

    public void receive( com.stc.connector.appconn.file.FileTextMessage
 input, dtd.otdOutputDTD1325973702.DB_Employee otdOutputDTD_DB_Employee_1,
 otdDB2Connect.OtdDB2ConnectOTD otdDB2Connect_1, com.stc.connector.
appconn.file.FileApplication FileClient_1 )
        throws Throwable
    {
        FileClient_1.setText( "Selectiong records from db_employee table
 via Table Select......." );
        FileClient_1.write();
        otdDB2Connect_1.getDb_employee().select( input.getText() );
        while (otdDB2Connect_1.getDb_employee().next()) {
            otdOutputDTD_DB_Employee_1.setEmpNo( typeConverter.shortToString(
 otdDB2Connect_1.getDb_employee().getEMP_NO(), "#", false, "" ) );
```

```
            otdOutputDTD_DB_Employee_1.setLastname( otdDB2Connect_1.
getDb_employee().getLAST_NAME() );
            otdOutputDTD_DB_Employee_1.setFirstname( otdDB2Connect_1.
getDb_employee().getFIRST_NAME() );
            otdOutputDTD_DB_Employee_1.setRate( otdDB2Connect_1.
getDb_employee().getRATE().toString() );
            otdOutputDTD_DB_Employee_1.setLastDate( typeConverter.
dateToString( otdDB2Connect_1.getDb_employee().getLAST_UPDATE(),
 "yyyy-MM-dd hh:mm:ss", false, "" ) );
            FileClient_1.setText( otdOutputDTD_DB_Employee_1.marshalToString() );
            FileClient_1.write();
        }
        FileClient_1.setText( "Table Select Done." );
        FileClient_1.write();
    }

}
```

The Update Operation

To perform an update operation on a table

1. **Execute the update() method.**

   Note – The content of the input.getText() file may contain null, meaning it will not have a "where" clause or it can contain a "where" clause such as empno > 50.

2. **Using a while loop together with next(), move to the row that you want to update.**

3. **Assign updating value(s) to the fields of the table OTD**

4. **Update the row by calling updateRow().**

```
package prjDB2Connect_JCDjcdALL;


public class jcdUpdate
{

public com.stc.codegen.logger.Logger logger;
public com.stc.codegen.alerter.Alerter alerter;
public com.stc.codegen.util.CollaborationContext collabContext;
public com.stc.codegen.util.TypeConverter typeConverter;

    public void receive( com.stc.connector.appconn.file.FileTextMessage
 input, otdDB2Connect.OtdDB2ConnectOTD otdDB2Connect_1, com.stc.connector.
appconn.file.FileApplication FileClient_1 )
        throws Throwable
    {
        FileClient_1.setText( "Updating the Rate and Last_update fields .. " );
        FileClient_1.write();
        otdDB2Connect_1.getDb_employee().update( input.getText() );
        while (otdDB2Connect_1.getDb_employee().next()) {
        otdDB2Connect_1.getDb_employee().setLAST_NAME( "Krishna" );
        otdDB2Connect_1.getDb_employee().setFIRST_NAME( "Kishore" );
        otdDB2Connect_1.getDb_employee().updateRow();
        }
        FileClient_1.setText( "Update Done." );
        FileClient_1.write();
    }

}
```

The Delete Operation

▼ To perform a delete operation on a table

● **Execute the delete() method.**

Note – The content of the input.getText() file may contain null, meaning it will not have a "where" clause or it can contain a "where" clause such as empno > 50.

In this example DELETE an employee.

```
package prjDB2Connect_JCDjcdALL;

public class jcdDelete
{
public com.stc.codegen.logger.Logger logger;
public com.stc.codegen.alerter.Alerter alerter;
public com.stc.codegen.util.CollaborationContext collabContext;
public com.stc.codegen.util.TypeConverter typeConverter;

    public void receive( com.stc.connector.appconn.file.FileTextMessage
 input, otdDB2Connect.OtdDB2ConnectOTD otdDB2Connect_1, com.stc.connector.
appconn.file.FileApplication FileClient_1 )
        throws Throwable
    {
        FileClient_1.setText( "Deleting record..........." );
        FileClient_1.write();
        otdDB2Connect_1.getDb_employee().delete( input.getText() );
        FileClient_1.setText( "Delete Done." );
        FileClient_1.write();
    }
}
```

Prepared Statement

A Prepared Statement is a SQL statement which can also contain parameter marker as input holder.

Example: select * from table1 where col1 > ?

This statement selects all the columns from a table called table1 if column col1 is greater than a certain value. The value will be supplied during runtime.

Note – The DB2 Connect Universal Driver only supports Updatable Resultsets for Update and Delete. The Insert operation is not supported. You can use a Prepared Statement to perform the Insert operation.

The Insert Operation

To perform an insert operation using Prepared Statement

1. Assign values to input fields.
2. Execute the executeUpdate().

This example inserts employee records. The Prepared Statement looks like this:

```
Insert into DB_EMPLOYEE values (?, ?, ?, ?, ?)
```

Note – If you don't want to insert values into all columns, your insert statement should look like this:

```
Insert into DB_EMPLOYEE (col1, col2) values (?, ?)

public class jcdPsInsert
{
    public com.stc.codegen.logger.Logger logger;

    public com.stc.codegen.alerter.Alerter alerter;

    public com.stc.codegen.util.CollaborationContext collabContext;

    public com.stc.codegen.util.TypeConverter typeConverter;
    public void receive( com.stc.connector.appconn.file.FileTextMessage input,
 otdDB2Connect.OtdDB2ConnectOTD otdDB2Connect_1, com.stc.connector.appconn.file.
FileApplication FileClient_1, dtd.otdInputDTD_654315252.DBemployees
otdInputDTD_DBemployees_1, dtd.otdOutputDTD1750519912.DBemployee
otdOutputDTD_DBemployee_1 )

        throws Throwable

    {

        FileClient_1.setText( "Inserting records into db_employee table using
Prepared Statement....." );

        FileClient_1.write();

        otdInputDTD_DBemployees_1.unmarshalFromString( input.getText() );

        for (int i1 = 0; i1 < otdInputDTD_DBemployees_1.countX_sequence_A(); i1 += 1) {

            otdDB2Connect_1.getInsert_ps().setEMP_NO( typeConverter.stringToShort(
otdInputDTD_DBemployees_1.getX_sequence_A( i1 ).getEmpNo(), "#", false, 0 ) );

            otdDB2Connect_1.getInsert_ps().setLAST_NAME( otdInputDTD_DBemployees_1.
getX_sequence_A( i1 ).getLastname() );

            otdDB2Connect_1.getInsert_ps().setFIRST_NAME( otdInputDTD_DBemployees_1.
getX_sequence_A( i1 ).getFirstname() );

            otdDB2Connect_1.getInsert_ps().setRATE( new java.math.BigDecimal(
otdInputDTD_DBemployees_1.getX_sequence_A( i1 ).getRate() ) );

            otdDB2Connect_1.getInsert_ps().setLAST_UPDATE( typeConverter.
stringToSQLDate( otdInputDTD_DBemployees_1.getX_sequence_A( i1 ).getLastDate(),
 "yyyy-MM-dd hh:mm:ss", false, "" ) );

            otdDB2Connect_1.getInsert_ps().executeUpdate();
        }

        FileClient_1.setText( "Insert Done......" );

        FileClient_1.write();

    }

}
```

The Update Operation

To perform an update operation using Prepared Statement

1. Assign value to input field.
2. Execute the executeUpdate() .

This example updates employee records which matches the where clause. The Prepared Statement looks like this:

Update DB_EMPLOYEE set rate = 19 where EMP_NO = ?

Note – The content of the input.getText() file must contain the input value to substitute the parameter marker ?

```
package prjDB2Connect_JCDjcdALL; public class jcdPsUpdate { public com.stc.codegen.logger.Logger logger; public
```

The Stored Procedure

A Stored Procedure OTD represents a database stored procedure. Fields correspond to the arguments of a stored procedure while methods are the operations that you can apply to the OTD. It allows you to execute a stored procedure. Remember that while in the Collaboration Editor you can drag and drop nodes from the OTD into the Collaboration Editor.

Executing Stored Procedures

The OTD represents the Stored Procedure “LookUpGlobal” with two parameters, an inbound parameter (INLOCALID) and an outbound parameter (OUTGLOBALPRODUCTID). These inbound and outbound parameters are generated by the DataBase Wizard and are represented in the resulting OTD as nodes. Within the Transformation Designer, you can drag values from the input parameters, execute the call, collect data, and drag the values to the output parameters.

Below are the steps for executing the Stored Procedure:

1. Specify the input values.
2. Execute the Stored Procedure.
3. Retrieve the output parameters if any.

For example:

```
package Storedprocedure;

public class sp_jce
{
    public com.stc.codegen.logger.Logger logger;

    public com.stc.codegen.alerter.Alerter alerter;

    public void receive( com.stc.connector.appconn.file.FileTextMessage
 input,com.stc.connector.appconn.file.FileApplication FileClient_1,
employeedb.Db_employee employeedb_with_top_db_employee_1,insert_DB.
Insert_DBOTD insert_DB_1 )
    throws Throwable
    {

        employeedb_with_top_db_employee_1.unmarshalFromString( input.getText() );

        insert_DB_1.getInsert_new_employee().setEmployee_no( java.lang.Integer.
parseInt( employeedb_with_top_db_employee_1.getEmployee_no() ) );

        insert_DB_1.getInsert_new_employee().setEmployee_Lname(
 employeedb_with_top_db_employee_1.getEmployee_lname() );

        insert_DB_1.getInsert_new_employee().setEmployee_Fname(
 employeedb_with_top_db_employee_1.getEmployee_fname() );

        insert_DB_1.getInsert_new_employee().setRate(
 java.lang.Float.parseFloat( employeedb_with_top_db_employee_1.getRate() ) );

        insert_DB_1.getInsert_new_employee().setUpdate_date(
 java.sql.Timestamp.valueOf( employeedb_with_top_db_employee_1.getUpdate_date() ) );

        insert_DB_1.getInsert_new_employee().execute();

        insert_DB_1.commit();

        FileClient_1.setText( "procedure executed" );

        FileClient_1.write();
    }

}
```

Manipulating the ResultSet and Update Count Returned by Stored Procedure

For Stored Procedures that return ResultSets and Update Count, the following methods are provided to manipulate the ResultSet:

- enableResultSetOnly
- enableUpdateCountsOnly

- enableResultSetandUpdateCounts
- resultsAvailable
- next
- getUpdateCount
- available

[Please define the ProductName text entity] stored procedures do not return records as ResultSets, instead, the records are returned through output reference cursor parameters. Reference Cursor parameters are essentially ResultSets.

The **resultsAvailable()** method, added to the PreparedStatementAgent class, simplifies the whole process of determining whether any results, be it Update Counts or ResultSets, are available after a stored procedure has been executed. Although JDBC provides three methods (**getMoreResults()**, **getUpdateCount()**, and **getResultSet()**) to access the results of a stored procedure call, the information returned from these methods can be quite confusing to the inexperienced Java JDBC programmer and they also differ between vendors. You can simply call **resultsAvailable()** and if Boolean true is returned, you can expect either a valid Update Count when **getUpdateCount()** is called and/or the next ResultSet has been retrieved and made available to one of the ResultSet nodes defined for the Stored Procedure OTD, when that node's **available()** method returns true.

Frequently, Update Counts information that is returned from a Stored Procedures is insignificant. You should process returned ResultSet information and avoid looping through all of the Update Counts. The following three methods control exactly what information should be returned from a stored procedure call. The **enableResultSetsOnly()** method, added to the PreparedStatement Agent class allows only ResultSets to be returned and thus every **resultsAvailable()** called only returns Boolean true if a ResultSet is available. Likewise, the **enableUpdateCountsOnly()** causes **resultsAvailable()** to return true only if an Update Count is available. The default case of **enableResultsetsAndUpdateCount()** method allows both ResultSets and Update Counts to be returned.

Collaboration usability for a stored procedure ResultSet

The Column data of the ResultSets can be dragged-and-dropped from their nodes to the Business Rules. Below is a code snippet that can be generated by the Collaboration Editor:

```
while (getSPIn().getSpS_multi().resultsAvailable())
{
if (getSPIn().getSpS_multi().getUpdateCount() > 0)
  {
    System.err.println("Updated "+getSPIn().getSpS_multi().getUpdateCount()+" rows");
  }

  if (getSPIn().getSpS_multi().getNormRS().available())
  {
    while (getSPIn().getSpS_multi().getNormRS().next())
    {
      System.err.println("Customer Id   = "+getSPIn().getSpS_multi().
getNormRS().getCustomerId());
      System.err.println("Customer Name = "+getSPIn().getSpS_multi().
getNormRS().getCustomerName());
      System.err.println();
    }
    System.err.println("===");
  }
  else if (getSPIn().getSpS_multi().getDbEmployee().available())
  {
    while (getSPIn().getSpS_multi().getDbEmployee().next())
    {
      System.err.println("EMPNO    = "+getSPIn().getSpS_multi().
getDbEmployee().getEMPNO());
      System.err.println("ENAME    = "+getSPIn().getSpS_multi().
getDbEmployee().getENAME());
      System.err.println("JOB      = "+getSPIn().getSpS_multi().
getDbEmployee().getJOB());
      System.err.println("MGR      = "+getSPIn().getSpS_multi().
getDbEmployee().getMGR());
      System.err.println("HIREDATE = "+getSPIn().getSpS_multi().
getDbEmployee().getHIREDATE());
      System.err.println("SAL      = "+getSPIn().getSpS_multi().
getDbEmployee().getSAL());
      System.err.println("COMM     = "+getSPIn().getSpS_multi().
getDbEmployee().getCOMM());
      System.err.println("DEPTNO   = "+getSPIn().getSpS_multi().
getDbEmployee().getDEPTNO());
      System.err.println();
    }
    System.err.println("===");
  }
}
```

Note – **resultsAvailable()** and **available()** cannot be indiscriminately called because each time they move ResultSet pointers to the appropriate locations.

After calling "**resultsAvailable()**", the next result (if available) can be either a **ResultSet** or an **UpdateCount** if the default "**enableResultSetsAndUpdateCount()**" was used.

Because of limitations imposed by some DBMSs, it is recommended that for maximum portability, all of the results in a ResultSet object should be retrieved before OUT parameters are retrieved. Therefore, you should retrieve all ResultSet(s) and Update Counts first followed by retrieving the OUT type parameters and return values.

The following list includes specific ResultSet behavior that you may encounter:

- The method **resultsAvailable()** implicitly calls **getMoreResults()** when it is called more than once. You should not call both methods in your java code. Doing so may result in skipped data from one of the ResultSets when more than one ResultSet is present.
- The methods **available()** and **getResultSet()** can not be used in conjunction with multiple ResultSets being open at the same time. Attempting to open more the one ResultSet at the same time closes the previous ResultSet. The recommended working pattern is:
  - Open one Result Set (ResultSet_1) and work with the data until you have completed your modifications and updates. Open ResultSet_2, (ResultSet_1 is now closed) and modify. When you have completed your work in ResultSet_2, open any additional ResultSets or close ResultSet_2.

  If you modify the ResultSet generated by the Execute mode of the Database Wizard, you need to assure the indexes match the stored procedure. By doing this, your ResultSet indexes are preserved.
- Generally, getMoreResults does not need to be called. It is needed if you do not want to use our enhanced methods and you want to follow the traditional JDBC calls on your own.

The DBWizard Assistant expects the column names to be in English when creating a ResultSet.

Prepared Statement

A Prepared Statement OTD represents a SQL statement that has been compiled. Fields in the OTD correspond to the input values that users need to provide.

Prepared statements can be used to perform insert, update, delete and query operations. A prepared statement uses a question mark (?) as a place holder for input. For example:

```
insert into EMP_TAB(Age, Name, Dept No) value(?, ?, ?)
```

To execute a prepared statement, set the input parameters and call **executeUpdate()** and specify the input values if any.

```
getPrepStatement().getPreparedStatementTest().setAge(23);
getPrepStatement().getPreparedStatementTest().setName(”Peter Pan’);
getPrepStatement().getPreparedStatementTest().setDeptNo(6);
getPrepStatement().getPreparedStatementTest().executeUpdate();
```

Batch Operations

To achieve better performance, consider using a bulk insert if you have to insert many records. This is the “Add Batch” capability. The only modification required is to include the **addBatch()** method for each SQL operation and then the **executeBatch()** call to submit the batch to the database server. Batch operations apply only to Prepared Statements.

```
getPrepStatement().getPreparedStatementTest().setAge(23);
getPrepStatement().getPreparedStatementTest().setName(”Peter Pan’);
getPrepStatement().getPreparedStatementTest().setDeptNo(6);
getPrepStatement().getPreparedStatementTest().addBatch();

getPrepStatement().getPreparedStatementTest().setAge(45);
getPrepStatement().getPreparedStatementTest().setName(”Harrison Ford’);
getPrepStatement().getPreparedStatementTest().setDeptNo(7);
getPrepStatement().getPreparedStatementTest().addBatch();
getPrepStatement().getPreparedStatementTest().executeBatch();
```

Using Oracle Operations

The database operations used in the Oracle Adapter are used to access the Oracle database. Database operations are either accessed through Activities in BPEL, or through methods called from a JCD Collaboration.

- “Oracle Adapter Database Operations (BPEL)” on page 76
- “Oracle Adapter Database Operations (JCD)” on page 79
- “Oracle Table Data Types” on page 88

Oracle Adapter Database Operations (BPEL)

The Oracle Adapter uses a number operations to query the Oracle database. Within a BPEL business process, the Oracle Adapter uses BPEL Activities to perform basic outbound database operations, including:

- Insert
- Update
- Delete
- SelectOne
- SelectMultiple
- SelectAll

In addition to these outbound operations, the Oracle Adapter also employs the inbound Activity **ReceiveOne** within a Prepared Statement OTD.

Activity Input and Output

The Sun SeeBeyond Enterprise Desinger– Business Rules Designer includes Input and Output columns to map and transform data between Activities displayed on the Business Process Canvas.

Figure 1–7 displays the business rules between the **FileClient.write** and **otdOracle.Db_employeeDelete** Activities. In this example, the **whereClause** appears on the Input side.

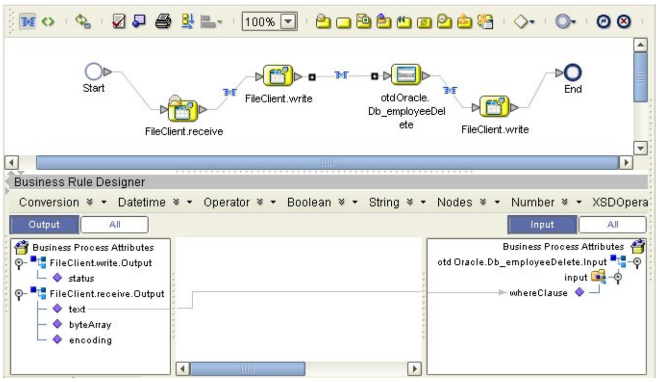


FIGURE 1–7 Input and Output Between Activities

The following table lists the expected Input and Output of each database operation Activity.

TABLE 1–7 Oracle Operations

eInsight Operations	Activity Input	Activity Output
SelectAll	where() clause (optional)	Returns all rows that fit the condition of the where() clause.
SelectMultiple	number of rows where() clause (optional)	Returns the number of rows specified that fit the condition of the where() clause, and the number of rows to be returned. For example: If the number of rows that meet the condition are 5 and the number of available rows are 10, then only 5 rows will be returned. Alternately, if the number of rows that meet the condition are 20, but if the number of available rows are 10, then only 10 rows are returned.
SelectOne	where() clause (optional)	Returns the first row that fits the condition of the where() clause.

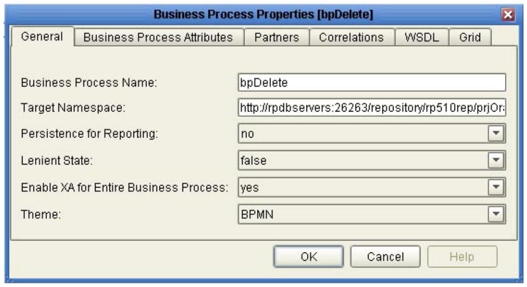
TABLE 1–7 Oracle Operations *(Continued)*

eInsight Operations	Activity Input	Activity Output
Insert	definition of new item to be inserted	Returns status.
Update	where() clause	Returns status.
Delete	where() clause	Returns status.

Oracle Adapter Outbound XA Support for BPEL

To enable XA support for BPEL in the Oracle Adapter do the following:

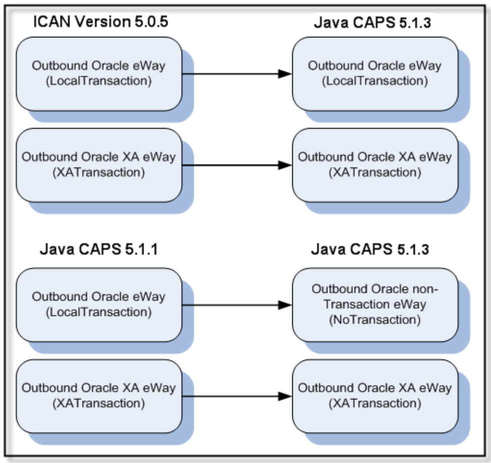
1. In the Business Process properties, set the Enable XA for the Entire Business Process field to Yes (see Figure 1–8).


Business Process Properties [bpDelete]
General
Business Process Attributes
Partners
Correlations
WSDL
Grid
OK
Cancel
Help

Business Process Name: bpDelete
Target Namespace: http://rpdbservers:26263/repository/rp510rep/prjOr
Persistence for Reporting: no
Lenient State: false
Enable XA for Entire Business Process: yes
Theme: BPMN

FIGURE 1–8 Business Process Properties

1. For all needed activities in the Business Process, set the Transaction Support field to Participates (see Figure 1–8).


ICAN Version 5.0.5
Java CAPS 5.1.3
Outbound Oracle eWay (LocalTransaction)
Outbound Oracle eWay (LocalTransaction)
Outbound Oracle XA eWay (XATransaction)
Outbound Oracle XA eWay (XATransaction)
Java CAPS 5.1.1
Java CAPS 5.1.3
Outbound Oracle eWay (LocalTransaction)
Outbound Oracle non-Transaction eWay (NoTransaction)
Outbound Oracle XA eWay (XATransaction)
Outbound Oracle XA eWay (XATransaction)

Note – For more information on XA support, refer to the Sun SeeBeyond eInsight Business Process Manager User's Guide.

Oracle Adapter Database Operations (JCD)

The same database operations are also used in the JCD, but appear as methods to call from the Collaboration.

Tables, Views, and Stored Procedures are manipulated through OTDs. Methods to call include:

- insert()
- insertRow()
- update(*String sWhere*)
- updateRow()
- delete(*String sWhere*)
- deleteRow()
- select(*String where*)

Note – Refer to the Javadoc for a full description of methods included in the Oracle Adapter.

The Table

A table OTD represents a database table. It consists of fields and methods. Fields correspond to the columns of a table while methods are the operations that you can apply to the OTD. This allows you to perform query, update, insert, and delete SQL operations in a table.

By default, the Table OTD has UpdatableConcurrency and ScrollTypeForwardOnly. The type of result returned by the select() method can be specified using:

- SetConcurrencytoUpdatable
- SetConcurrencytoReadOnly
- SetScrollTypetoForwardOnly
- SetScrollTypetoScrollSensitive
- SetScrollTypetoInsensitive

The Query (Select) Operation

▼ To perform a query operation on a table

1 Execute the select() method with the "where" clause specified if necessary.

Note – The content of the input.getText() file may contain null, meaning it will not have a "where" clause or it can contain a "where" clause such as empno > 50.

---

2 Loop through the ResultSet using the next() method.

3 Process the return record within a while() loop.

For example:

```
package prjOracle_JCDjcdALL;


public class jcdTableSelect
{

    public com.stc.codegen.logger.Logger logger;

    public com.stc.codegen.alerter.Alerter alerter;

    public com.stc.codegen.util.CollaborationContext collabContext;

    public com.stc.codegen.util.TypeConverter typeConverter;

    public void receive( com.stc.connector.appconn.file.FileTextMessage
 input, dtd.otdOutputDTD1325973702.DB_Employee otdOutputDTD_DB_Employee_1,
 otdOracle.OtdOracleOTD otdOracle_1, com.stc.connector.appconn.file.
FileApplication FileClient_1 )
        throws Throwable
    {
        FileClient_1.setText( "Selectiong records from db_employee table via
 Table Select........" );
        FileClient_1.write();
        otdOracle_1.getDb_employee().select( input.getText() );
        while (otdOracle_1.getDb_employee().next()) {
            otdOutputDTD_DB_Employee_1.setEmpNo( typeConverter.shortToString(
 otdOracle_1.getDb_employee().getEMP_NO(), "#", false, "" ) );
            otdOutputDTD_DB_Employee_1.setLastname( otdOracle_1.getDb_employee().
getLAST_NAME() );
            otdOutputDTD_DB_Employee_1.setFirstname( otdOracle_1.getDb_employee().
getFIRST_NAME() );
            otdOutputDTD_DB_Employee_1.setRate( otdOracle_1.getDb_employee().
getRATE().toString() );
            otdOutputDTD_DB_Employee_1.setLastDate( typeConverter.dateToString(
 otdOracle_1.getDb_employee().getLAST_UPDATE(), "yyyy-MM-dd hh:mm:ss", false, "" ) );
            FileClient_1.setText( otdOutputDTD_DB_Employee_1.marshalToString() );
            FileClient_1.write();
        }
        FileClient_1.setText( "Table Select Done." );
        FileClient_1.write();
    }

}
```

The Insert Operation

▼ To perform an insert operation on a table

1. **Execute the insert() method. Assign a field.**

2. **Insert the row by calling insertRow()**

   This example inserts an employee record.

```
package prjOracle_JCDjcdALL;


public class jcdInsert
{

    public com.stc.codegen.logger.Logger logger;

    public com.stc.codegen.alerter.Alerter alerter;
    public com.stc.codegen.util.CollaborationContext collabContext;

    public com.stc.codegen.util.TypeConverter typeConverter;

    public void receive( com.stc.connector.appconn.file.FileTextMessage input,
 otdOracle.OtdOracleOTD otdOracle_1,     dtd.otdInputDTD_1206505729.DB_Employee
 otdInputDTD_DB_Employee_1, com.stc.connector.appconn.file.FileApplication
FileClient_1 ) throws Throwable
    {
        FileClient_1.setText( "Inserting records in to db_employee table......" );
        FileClient_1.write();
        otdInputDTD_DB_Employee_1.unmarshalFromString( input.getText() );
        otdOracle_1.getDb_employee().insert();
        for (int i1 = 0; i1 < otdInputDTD_DB_Employee_1.countX_sequence_A(); i1 += 1) {
            otdOracle_1.getDb_employee().setEMP_NO( typeConverter.stringToShort(
 otdInputDTD_DB_Employee_1.getX_sequence_A( i1 ).getEmpNo(), "#", false, 0 ) );
            otdOracle_1.getDb_employee().setLAST_NAME( otdInputDTD_DB_Employee_1.
getX_sequence_A( i1 ).getLastname() );
            otdOracle_1.getDb_employee().setFIRST_NAME( otdInputDTD_DB_Employee_1.
getX_sequence_A( i1 ).getFirstname() );
            otdOracle_1.getDb_employee().setRATE( new java.math.BigDecimal(
 otdInputDTD_DB_Employee_1.getX_sequence_A( i1 ).getRate() ) );
            otdOracle_1.getDb_employee().setLAST_UPDATE( typeConverter.
stringToTimestamp( otdInputDTD_DB_Employee_1.getX_sequence_A( i1 ).getLastDate(),
 "yyyy-MM-dd hh:mm:ss", false, "" ) );
            otdOracle_1.getDb_employee().insertRow();
        }
        FileClient_1.setText( "Insert Done." );
        FileClient_1.write();
    }

}
```

The Update Operation

▼ To perform an update operation on a table

1. **Execute the update() method.**

   Note – The content of the input.getText() file may contain null, meaning it will not have a “where” clause or it can contain a “where” clause such as empno > 50.

2. **Using a while loop together with next(), move to the row that you want to update.**

3. **Assign updating value(s) to the fields of the table OTD**

4 **Update the row by calling updateRow().**

```
package prjOracle_JCDjcdALL;


public class jcdUpdate
{

    public com.stc.codegen.logger.Logger logger;

    public com.stc.codegen.alerter.Alerter alerter;

    public com.stc.codegen.util.CollaborationContext collabContext;

    public com.stc.codegen.util.TypeConverter typeConverter;

    public void receive( com.stc.connector.appconn.file.FileTextMessage input,
 otdOracle.OtdOracleOTD otdOracle_1, com.stc.connector.appconn.file.FileApplication
 FileClient_1 ) throws Throwable
    {
        FileClient_1.setText( "Updating the Rate and Last_update fields .. " );
        FileClient_1.write();
        otdOracle_1.getDb_employee().update( input.getText() );
        while (otdOracle_1.getDb_employee().next()) {
        otdOracle_1.getDb_employee().setLAST_NAME( "Krishna" );
        otdOracle_1.getDb_employee().setFIRST_NAME( "Kishore" );
        otdOracle_1.getDb_employee().updateRow();
        }
        FileClient_1.setText( "Update Done." );
        FileClient_1.write();
    }

}
```

The Delete Operation

▼ To perform a delete operation on a table

● **Execute the delete() method.**

Note – The content of the input.getText() file may contain null, meaning it will not have a “where” clause or it can contain a “where” clause such as empno > 50.

In this example DELETE an employee.

```
package prjOracle_JCDjcdALL;

public class jcdDelete
{
    public com.stc.codegen.logger.Logger logger;

    public com.stc.codegen.alerter.Alerter alerter;

    public com.stc.codegen.util.CollaborationContext collabContext;

    public com.stc.codegen.util.TypeConverter typeConverter;

    public void receive( com.stc.connector.appconn.file.FileTextMessage input,
 otdOracle.OtdOracleOTD otdOracle_1, com.stc.connector.appconn.file.FileApplication
 FileClient_1 ) throws Throwable
    {
        FileClient_1.setText( "Deleting record..........." );
        FileClient_1.write();
        otdOracle_1.getDb_employee().delete( input.getText() );
        FileClient_1.setText( "Delete Done." );
        FileClient_1.write();
    }
}
```

The Stored Procedure

A Stored Procedure OTD represents a database stored procedure. Fields correspond to the arguments of a stored procedure while methods are the operations that you can apply to the OTD. It allows you to execute a stored procedure. In the Collaboration Editor you can assign values to the input parameters, execute the call, collect data, and retrieve the values from output parameters.

Executing Stored Procedures

The OTD used in the example below, contains a Stored Procedure with input parameters. These input parameters are generated by the Database OTD Wizard and are displayed in the Collaboration Editor as subnodes of the OTD.

Below are the steps for executing the Stored Procedure:

1. Specify the input values.
2. Execute the Stored Procedure.
3. Retrieve the output parameters if any.

For example:

```
package Storedprocedure;

public class sp_jce
{

    public com.stc.codegen.logger.Logger logger;

    public com.stc.codegen.alerter.Alerter alerter;

    public void receive( com.stc.connector.appconn.file.FileTextMessage
 input,com.stc.connector.appconn.file.FileApplication FileClient_1,employeedb.
Db_employee employeedb_with_top_db_employee_1,insert_DB.Insert_DBOTD insert_DB_1 )
    throws Throwable
    {
        employeedb_with_top_db_employee_1.unmarshalFromString( input.getText() );

insert_DB_1.getInsert_new_employee().setEmployee_no( java.lang.Integer.parseInt(
 employeedb_with_top_db_employee_1.getEmployee_no() ) );

        insert_DB_1.getInsert_new_employee().setEmployee_Lname(
 employeedb_with_top_db_employee_1.getEmployee_lname() );

        insert_DB_1.getInsert_new_employee().setEmployee_Fname(
 employeedb_with_top_db_employee_1.getEmployee_fname() );

        insert_DB_1.getInsert_new_employee().setRate(
 java.lang.Float.parseFloat( employeedb_with_top_db_employee_1.getRate() ) );

        insert_DB_1.getInsert_new_employee().setUpdate_date(
 java.sql.Timestamp.valueOf( employeedb_with_top_db_employee_1.getUpdate_date() ) );

        insert_DB_1.getInsert_new_employee().execute();

        insert_DB_1.commit();

        FileClient_1.setText( "procedure executed" );

        FileClient_1.write();
    }

}
```

Manipulating the ResultSet and Update Count Returned by Stored Procedure

The following methods are provided for using the ResultSet and Update Count when they are returned by Stored Procedures:

- enableResultSetOnly

- enableUpdateCountsOnly
- enableResultSetandUpdateCounts
- resultsAvailable
- next
- getUpdateCount
- available

Note – Stored Procedure ResultSets are supported in Java collaborations only.

Oracle stored procedures do not return records as ResultSets; instead, the records are returned through output reference cursor parameters. Reference Cursor parameters are essentially ResultSets.

The resultsAvailable() method, added to the OTD, simplifies the whole process of determining whether any results, whether they are update Counts or ResultSets, are available after a stored procedure has been executed. Although JDBC provides three methods (getMoreResults(), getUpdateCount(), and getResultSet()) to access the results of a stored procedure call, the information returned from these methods can be quite confusing to the inexperienced Java JDBC programmer and they also differ between vendors. You can simply call resultsAvailable() and if Boolean true is returned, you can expect either a valid Update Count when getUpdateCount() is called and/or the next ResultSet has been retrieved and made available to one of the ResultSet nodes defined for the Stored Procedure OTD, when that node's available() method returns true.

Update Counts information that is returned from Stored Procedures is often insignificant. Process returned ResultSet information and avoid looping through all of the Update Counts. The following three methods control exactly what information is returned from a stored procedure call. The enableResultSetsOnly() method, added to the OTD allows only ResultSets to be returned and thus every resultsAvailable() called only returns Boolean true if a ResultSet is available. Likewise, the enableUpdateCountsOnly() method causes resultsAvailable() to return true only if an Update Count is available. The default case of the enableResultSetsAndUpdateCount() method enables both ResultSets and Update Counts to be returned.

Collaboration usability for a Stored Procedure ResultSet

You can use your mouse to drag and drop the Column data of the ResultSets from their OTD nodes to the Business Rules. Below is a code snippet that can be generated by the Collaboration Editor:

```
// resultsAvailable() true if there's an update count and/or a result set available.
// note, it should not be called indiscriminantly because each time the results
pointer is advanced via getMoreResults() call.
```

```
while (getSPIn().getSpS_multi().resultsAvailable())
{
    // check if there’s an update count
    if (getSPIn().getSpS_multi().getUpdateCount() > 0)
    {
        logger.info("Updated "+getSPIn().getSpS_multi().getUpdateCount()+" rows");
    }
// each result set node has an available() method (similar to OTD’s) that tells the
// user whether this particular result set is available. note, JDBC does support access
// to more than one result set at a time, i.e., cannot drag from two distinct result
// sets simultaneously
    if (getSPIn().getSpS_multi().getNormRS().available())
    {
    while (getSPIn().getSpS_multi().getNormRS().next())
    {
    logger.info("Customer Id = "+getSPIn().getSpS_multi().getNormRS().getCustomerId());
    logger.info("Customer Name = "+getSPIn().getSpS_multi().getNormRS().
getCustomerName());
    }
    if (getSPIn().getSpS_multi().getDbEmployee().available())
    {
    while (getSPIn().getSpS_multi().getDbEmployee().next())
    {
    logger.info("EMPNO = "+getSPIn().getSpS_multi().getDbEmployee().getEMPNO());
    logger.info("ENAME = "+getSPIn().getSpS_multi().getDbEmployee().getENAME());
    logger.info("JOB = "+getSPIn().getSpS_multi().getDbEmployee().getJOB());
    logger.info("MGR = "+getSPIn().getSpS_multi().getDbEmployee().getMGR());
    logger.info("HIREDATE = "+getSPIn().getSpS_multi().getDbEmployee().getHIREDATE());
    logger.info("SAL = "+getSPIn().getSpS_multi().getDbEmployee().getSAL());
    logger.info("COMM = "+getSPIn().getSpS_multi().getDbEmployee().getCOMM());
    logger.info("DEPTNO = "+getSPIn().getSpS_multi().getDbEmployee().getDEPTNO());
    }
}
```

Note – resultsAvailable() and available() cannot be indiscriminately called because each time they move ResultSet pointers to the appropriate locations.

Once the "resultsAvailable()" method has been called, the next result (if available) can be either a ResultSet or an UpdateCount, if the default "enableResultSetsAndUpdateCount()" was used.

Because of limitations imposed by some DBMSs, SeeBeyond recommends that for maximum portability, all of the results in a ResultSet object should be retrieved before OUT parameters are retrieved. Therefore, you must retrieve all ResultSet(s) and update counts first, followed by retrieving the OUT type parameters and return values.

The following list includes specific ResultSet behavior that you may encounter:

- The method resultsAvailable() implicitly calls getMoreResults() when it is called more than once. Do not call both methods in your Java code. If you do, there may be skipped data from one of the ResultSets when more than one ResultSet is present.
- The methods available() and getResultSet() cannot be used when mulitple ResultSets are open at the same time. Attempting to open more the one ResultSet at the same time closes the previous ResultSet. The recommended working pattern is:
  - Open one Result Set, ResultSet_1 and work with the data until you have completed your modifications and updates. Open ResultSet_2, (ResultSet_1 is now closed) and modify. When you have completed your work in ResultSet_2, open any additional ResultSets or close ResultSet_2.
- If you modify the ResultSet generated by the Execute mode of the Database Wizard, you need to make sure that the indexes match the stored procedure; if you do this, your ResultSet indexes are preserved.
- Generally, you do not need to call getMoreResults; you need to call it only if you do not want to use our enhanced methods and you want to follow the traditional JDBC calls on your own.

Oracle Table Data Types

Oracle tables support the following data types:

- Real - an approximate numeric data type.
- Float - a data type where all platforms have values of the least specified minimum precision.
- CLOB - a built-in data type that stores a Character Large Object as a column value in a row of a database table.

For all others, use the data types Float, Double, or CLOB and build them using a data type of “Other”.

Note – The Oracle driver does not support the boolean and PL/SQL RECORD datatypes in the Function and Stored Procedure.

Long RAW for Prepared Statements and Stored Procedure support:

The following two parameters must be set prior to the Insert/Update/Delete statement.

```
setConcurrencyToReadOnly()
setScrollTypeToForwardOnly()
```

Using CLOBs

To use a CLOB in the Oracle Adapter, do the following:

1. In the Enterprise Designer, right-click on the project, select Import. From the submenu, select **File**. The Import File dialog box appears.
2. From

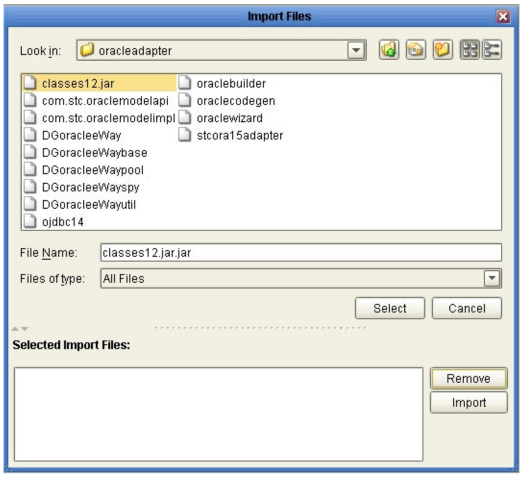
```
 <Client_eDesigner>\usrdir\modules\ext\oracleadapter, create a copy of the
 ojdbc14.jar
 in the directory and rename it
 classes12.jar
```

.

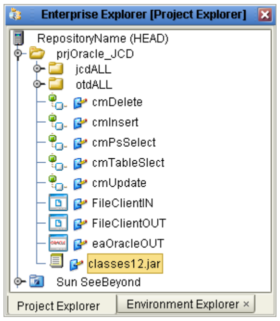
3. Navigate to the **classes12.jar file**,

```
 <Client_eDesigner>\usrdir\modules\ext\oracleadapter\
classes12.jar
 using the Enterprise Designer's Project File Import feature (see
```

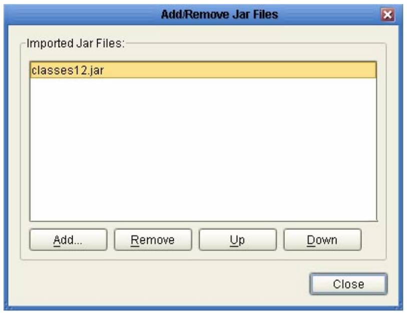
"Using CLOBs" on page 89



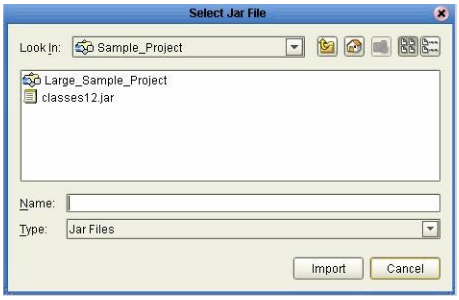
4. Click **Select**.
5. Click **Import**. The classes12.jar file appears, as shown in "Using CLOBs" on page 89.



6. To load the **classes12.jar** file into your Java Collaboration, select the **Import JAR File** button. Click Add in the Add/Remove Jar Files window to add the Jar files (see "Using CLOBs" on page 89).

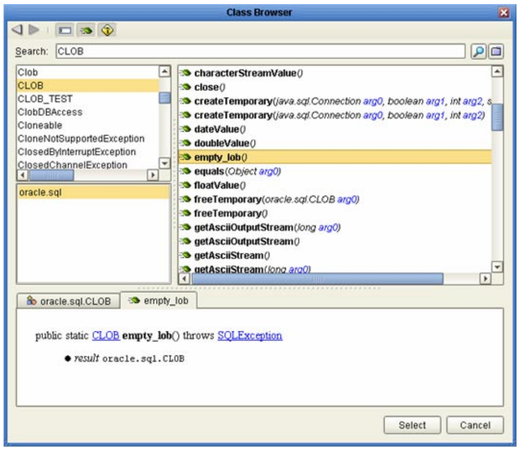


7. In the **Select Jar File** window, select the **classes12.jar** file and click **Import** (see "Using CLOBs" on page 89).



8. In the **Add/Remove Jar Files** window, click **Close**.

9. In the Business Rules Designer, call the CLOB method by clicking the Class Browser button. The Class Browser dialog box appears (see "Using CLOBs" on page 89).



Clob
CLOB
CLOB_TEST
ClobDBAccess
Cloneable
CloneNotSupportedException
ClosedByInterruptException
ClosedChannelException

oracle.sql

characterStreamValue()
close()
createTemporary(java.sql.Connection arg0, boolean arg1, int arg2, s
createTemporary(java.sql.Connection arg0, boolean arg1, int arg2)
dateValue()
doubleValue()
empty_lob()
equals(Object arg0)
floatValue()
freeTemporary(oracle.sql.CLOB arg0)
freeTemporary()
getAsciiOutputStream(long arg0)
getAsciiOutputStream()
getAsciiStream()
getAsciiStream(long arg0)

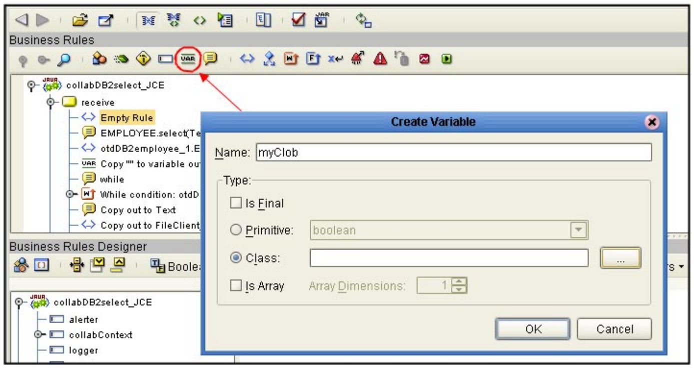
oracle.sql.CLOB | empty_lob

public static CLOB empty_lob() throws SQLException

- *result* oracle.sql.CLOB

10. Select empty_lob from the list of CLOB variables and click Select.

11. Create a local variable by clicking the Local Variable button on the Business Rules toolbar. The Create Variable dialog box is displayed. (see "Using CLOBs" on page 89).



collabDB2select_JCE
receive
Empty Rule
EMPLOYEE.select(Te
otdDB2employee_1.E
Copy "" to variable ou
while
While condition: otdD
Copy out to Text
Copy out to FileClient

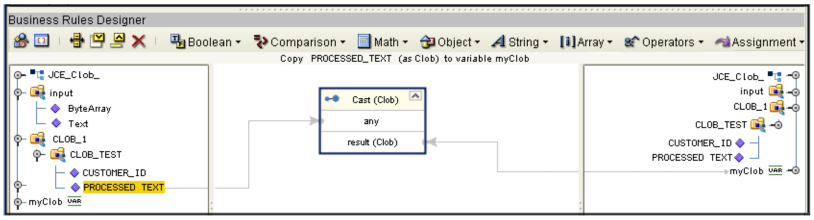
collabDB2select_JCE
alerter
collabContext
logger

Name: myClob
Type:
Is Final
Primitive: boolean
Class:
Is Array Array Dimensions: 1

12. Name the variable myCLOB, select the Class type, and choose CLOB as the Class type.

13. Click OK to create the variable.

14. In the Business Rules Designer, drag the CLOB to the Local Variable using the Cast method. Click Yes when the incompatible Data Type warning appears (see "Using CLOBs" on page 89).



Copy PROCESSED_TEXT (as Clob) to variable myClob

15. Use the CLOB putString method to assign 1 to Arg().
16. In the Java Collaboration Editor, the Java code resembles the following:

```
public void receive( com.stc.connector.appconn.file.FileTextMessage input,cLOB.
CLOBOTD CLOB_1 ) throws Throwable
{
    //@map:CLOB_1.getCLOB_TEST.insert
        CLOB_1.getCLOB_TEST().insert();

        //@map:Copy java.math.BigDecimal.valueOf(100) to CUSTOMER_ID
        CLOB_1.getCLOB_TEST().setCUSTOMER_ID( java.math.BigDecimal.
valueOf( 100 ) );

        //@map:Copy oracle.sql.CLOB.empty_lob to PROCESSED_TEXT
        CLOB_1.getCLOB_TEST().setPROCESSED_TEXT( oracle.sql.CLOB.empty_lob() );

        //@map:CLOB_TEST.insertRow
        CLOB_1.getCLOB_TEST().insertRow();

//@map:CLOB_1.getCLOB_TEST.select("customer_id = 100 for update")
        CLOB_1.getCLOB_TEST().select( "customer_id = 100 for update" );
//If
        if (CLOB_1.getCLOB_TEST().next()) {
            //@map:oracle.sql.CLOB myClob;
            oracle.sql.CLOB myClob;

            //@map:Copy cast PROCESSED_TEXT to oracle.sql.CLOB to myClob
            myClob = (oracle.sql.CLOB) CLOB_1.getCLOB_TEST().getPROCESSED_TEXT();

           //@map:myClob.putString(1,Text)
            myClob.putString( 1,input.getText() );

            //@map:CLOB_TEST.updateRow
            CLOB_1.getCLOB_TEST().updateRow();
}
    }
```