Deltek

Deltek Costpoint® 8.0

Extensibility Designer Coding Guide

September 1, 2020

While Deltek has attempted to verify that the information in this document is accurate and complete, some typographical or technical errors may exist. The recipient of this document is solely responsible for all decisions relating to or use of the information provided herein.

The information contained in this publication is effective as of the publication date below and is subject to change without notice.

This edition published September 2020.

Contents

Chapter 1: Standards 1
Java Standards 1
Package Structure 1
Class 2
Method 3
Variable 3
Constant 3
Comments 4
Indentation 4
Java Logging 4
Imports 4
Chapter 2: system.applicationinterface Package 5
system.applicationinterface.DEException 5
system.applicationinterface.ResultSetInterface 5
system.applicationinterface.RowSetInterface 8
system.applicationinterface.AppInterface 8
Global Constants 9
List of Global Constants 10
Using Constants as Variable in the Designer 10
Automatic Constant 10
Creating your own application constants 11
system.applicationinterface.LoggerInterface 12
How to Add Logging in Your Class: 13
CPlog4j.properties 13
system.applicationinterface.CPConstants 13
system.applicationinterface.SqlManager 14
Rules 14
Using DbF functions 16
Using other key words 16
system.applicationinterface.FileHandlerInterface 17
File Uploading 17
File Uploading UI 18
Alternate location 18

FileHandlerInterface 18
Chapter 3: system.utils Package 21
system.utils.Numbers 21
System.utils.UIFormat 21
Chapter 4: Plug In Events 23
Plug-in Classes Are Stateless 23
RS Population Event 23
RS Populate Event 23
Caching Scheme 23
Validation event 25
Types of Validation Events 25
Rowset definition 25
Object Validation 25
Row Validation 27
RS Validation 29
Connection Mode Summary 30
Validations Handled by System 31
Before & After Save 31
BeforeRSSave 32
AfterRSSave 33
Order on Save 35
Validation sequence (intra result set) 35
Validation sequence (inter result sets) 36
Save sequence 36
Actions 36
ActionInterface 36
Web Services 37
Extending a Standard Deltek Action 37
Adding a new extensibility action 39
Action Progress Meter 41
Maintenance Action 43

Chapter 1: Standards

Standards are necessary for ease of reference, maintenance and collaboration. This section provides standards to be followed for Java coding for Costpoint application and extensibility.

Java Standards

Costpoint development adopts the Java Naming Standards as the starting point for all coding conventions and standards.

> **Attention:** For more information, visit the following web page:
>
> http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html

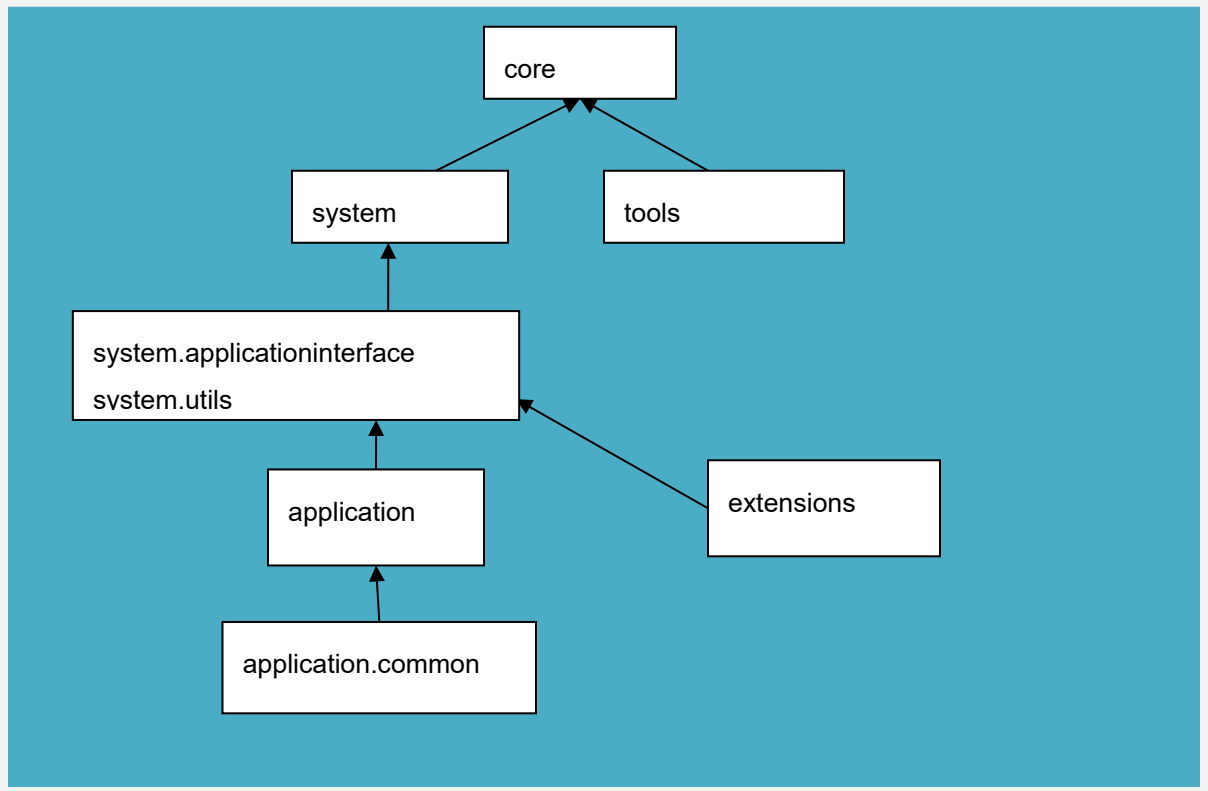
This section provides specific details that apply to Costpoint development. If the details are absent in this document, Java Naming Standards should be followed.

Package Structure

All package names are in lower case.

All Costpoint packages start with com.deltek.enterprise. There are four top branches underneath. They are **core**, **system**, **tools** and **application**. Packages for your extensibility will form the fifth branch named **extensions**.

Package	Description
com.deltek.enterprise.core	Private framework code. Extensibility developer may not use any of these packages.
com.deltek.enterprise.system	Private framework code except two packages in this branch that extensibility developer can use: applicationinterface and utils (see details later).
com.deltek.enterprise.tools	Private framework code. Extensibility developer may not use any of these packages.
com.deltek.enterprise.application	Private common or specific application code containing business rules. Extensibility developer may use this package.
com.deltek.enterprise.extensions	All code developed for application extensibility must reside under this package.



The diagram above shows that code in the extensions branch should only use packages from system.applicationinterface, and system.utils (not shown).

com.deltek.enterprise.extensions.<extensibility project id>

Under extensions package, the next level must be named the same as the extensiblity project ID (created in the Extensibility Designer)

For example

com.deltek.enterprise.extensions.xt_project_y

We only enforce the structure up to this level.

Under the project level, we recommend you sub package this further into branches for ease of reference.

For example

- com.deltek.enterprise.extensions.xt_project_y.common (to be shared by all code under this project)
- com.deltek.enterprise.extensions.xt_project_y.unit1.common (to be shared by all code under unit1)
- com.deltek.enterprise.extensions.xt_project_y.unit1.symusr (specific code for extending the SYMUSR app under unit 1)

Class

The class name should always start with an upper case letter. Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Keep your class names simple and descriptive. Do not use hyphen for class name.

For app specific java classes, developers should follow the below naming pattern:

<App ID> [Optional ChildRSName]<eventType>

Where **eventType** can be **RsPolulate**, **ObjValidation**, **LineValidation**, **RSValidation**, **BeforeSave**, or **AfterSave**. (see event plug in section).

For example:

- com.deltek.enterprise.extensions.xt_project_y.unit1.symusr.SymusrRsPopulate.java
- com.deltek.enterprise.extensions.xt_project_y.unit1.symusr.SymusrObjValidation.java

App common helper file can be named as <App ID>common.

For example:

- com.deltek.enterprise.extensions.xt_project_y.unit1.symusr.SymusrCommon.java

Method

Methods should be verbs, in mixed case with the first letter lowercase and the first letter of each internal word capitalized.

For example

- validateHdrInfo
- computeExchgRates

Variable

Variable names should be short yet meaningful. The choice of a variable name should be mnemonic- that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary "throwaway" variables.

Similar to method name, variable names are in mixed case with the first letter lowercase and the first letter of each internal word capitalized. Do not use underscores.

Variable holding value from a database column should assume the name of the database column (without the underscores).

For example:

- projId for column PROJ_ID
- vchrKey for column VCHR_KEY

Constant

The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores ("_").

For example:

- public static final short OPEN_APP
- public static final short CLOSE_APP

Comments

Comments for classes and for methods should follow the Javadoc guidelines.

> **Attention:** For more information, please refer to the following site:
>
> http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

Indentation

All source code should be properly indented for ease of reading.

Java Logging

Avoid using System.out and use LoggerInterface to enable more flexibility and extensibility in logging information.

> **Attention:** For more information, please refer to the LoggerInterface section.

Imports

When developing application java code, only two packages can be imported from the system branch:

- import com.deltek.enterprise.system.applicationinterface.*;
- import com.deltek.enterprise.system.utils.*;

You can import any packages from any common code that you **create for your extensions packages**.

- import com.deltek.enterprise.extensions.xt_project_y.common.*
- import java.util.*
- import java.lang.*
- import java.text.*

You can import these standard Java packages if you need them for development.

- import java.sql.SQLException;

Import this class if your class accesses the database (via SqlManager class – discussed later)

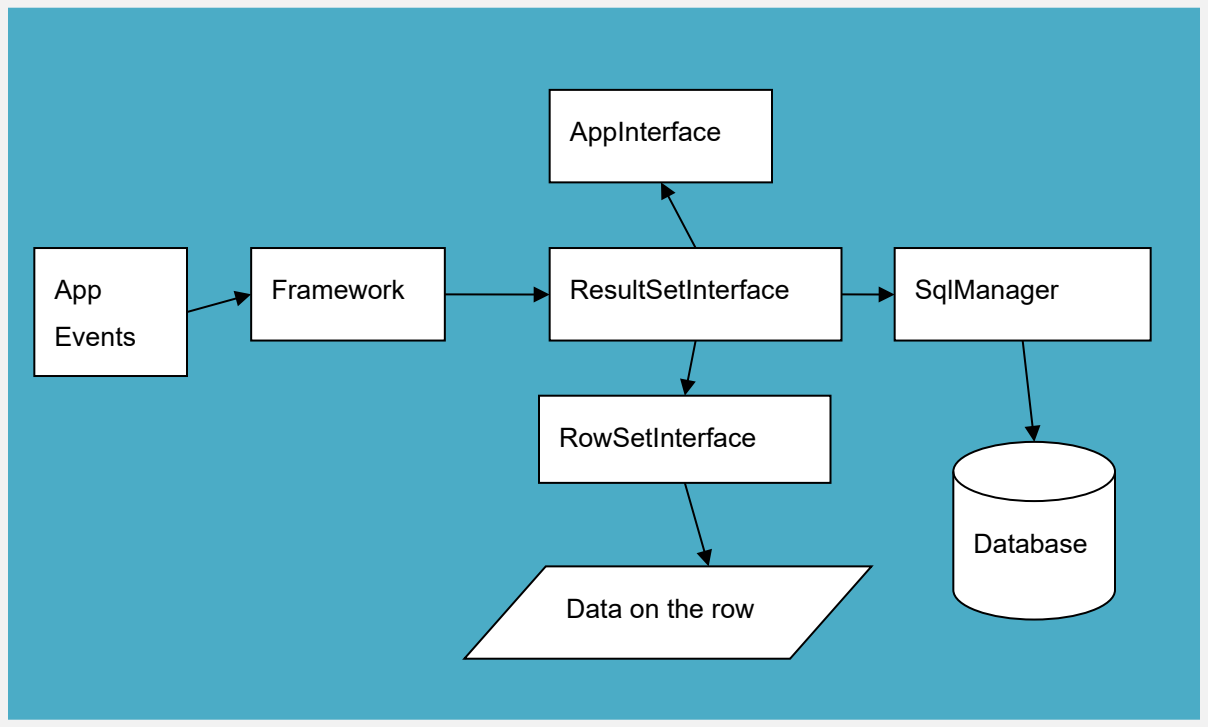
Chapter 2: system.applicationinterface Package

Upon certain events triggered by the user, the framework will call the plug-in java classes registered by the developer in the Designer tool for that event.

For example, on Query, the framework will call the App Populate event plug-in. On tabbing out of a field (and user login with Field mode), the framework will call object validation event plug-in. On leaving a line to the next line (and user login with Field mode or Line mode), the framework will call line validation plug-in event (See plug-in event section for more details).

When calling the plug-in method, the framework will pass the handle to current screen (result set) the ResultSetInterface. From this handle, application code can access and manipuldate data contain in that result set and perform calculation or validation as necessary. Data on the screen can be manipulated via the RowsetInterface while data in the database can be directly updated via the SqlManager interface.

> **Note:** There are exceptions when the framework passes AppInterface and ActionInterface (discussed later).



system.applicationinterface.DEException

This class is a wrapper class for the root class java.lang.Exception. All application method must throw this exception instead of various types of java exception. The system will catch this exception and handle it gracefully without bringing down the entire session or even the application server.

system.applicationinterface.ResultSetInterface

ResultSetInterface is the most common interface between the system and the application java classes.

This interface is passed to the result set each time a plug-in method is called from the system.

> **Note:** In some instances, appInterface or actionInterface is passed instead – explained later.

This interface provides methods to get to objects on the current result set, the parent's or children result set.

Here are some commonly used methods provided by the interface (see javadoc for more explanations)

- **getRowSet()**: Returns the RowSetInterface to the row in context. You always need to get RowSetInterface before you can get to the data on the row.
- **getApplication()**: Returns the AppInterface to the app user is in.
- **getSqlManager(Object o)**: Returns an instance of SqlManager which is used to access the database via SQL.
- **getParentRS()**: Returns the ResultSetInterface of the parent RS when the application contains more than one result set. Always check for null before using the returned ResultSetInterface (if user does not open the subtask screen, ResultSetInterface of the subtask may be null).
- **getChild(String rsId)**: Returns the ResultSetInterface of the specified child RS. Always check for null before using the returned ResultSetInterface.
- **findInit(int flagsOn,int flagsOff,boolean setContext)**: Returns an iterator so you can loop through rows in the result set.
- **findFirstRow(int flagsOn,int flagsOff)**: Returns the number of the first row (each row has a number) found in the search.
- **addObjectMessage(String objectId,String msgId, short msgType)**: Add a message and its severity. Set focus to the indicated object in the UI.

  Overload method:

  addObjectMessage(String objectId,String msgId, short msgType, String parms[]): Add a tokenized message and its severity and associate with an object.
- **addLineMessage(String msgId, short msgType)**: Add a message and its severity. Set focus to the entire line in the UI.

Example: Validate a project ID and issue a error message for invalid project id

```
/*****************************************************************************/
package com.deltek.enterprise.extensions.xt_project_y.mypjmbasic;

import com.deltek.enterprise.system.applicationinterface.DEException;
import com.deltek.enterprise.system.applicationinterface.CPConstants;
import com.deltek.enterprise.system.applicationinterface.ResultSetInterface;
import com.deltek.enterprise.system.applicationinterface.RowSetInterface;


public class MyPjmbasicObjectValidation {
   public short validateProjId (ResultSetInterface rsI) throws DEException {
       /* Initialize return code to OK */
```

```
        short retCode = CPConstants.CP_OK;

        /* Get to RowSetInterface from ResultSetInterface */
        RowSetInterface roI = rsI.getRowSet();

        /* Get data on the row */
        String projID = roI.getStringValue("PROJ_ID");

        /* Call some method to validate the value of project ID */
        if (!checkProjID(projID))  {
                /* Return an error if project ID is not validated */
                rsI.addObjectMessage("PROJ_ID","CP_PROJ_INVALID", rsI.ERROR);
                retCode = rsI.ERROR;
        }
        return retCode;
    }
}
/*************************************************************************/
```

Message severity

```
INFORMATION = 1 = Info message only.
WARNING = 2 = Warning (OK/Cancel)
ERROR = 3 = Error (Error but continue validation to find more error)
FATAL = 4 = Fatal (Error and stop validation)
```

Message methods

- **addObjectMessage (String objectId, String msgId, short severityCode)**: Return a message with the severity code and set focus on the object Id specified when user clicks on the message link on the UI. ObjectID is the object ID specified in the Designer. Message ID identifies the message stored in the message table in the database.

  > **Attention:** See Javadoc for additional overloading addObjectMessage methods that passes additional parameters.

- **addLineMessage (String msgId, short sevType)**: Same as addObjectMessage but return focus to the entire line when user clicks on the message link on the UI
- **addRSMessage (String msgId, short sevType)**: Same as addObjectMessage but return focus to the entire result set when user clicks on the message link on the UI

Example: Looping through result set

```
/*************************************************************************/
private void loopMyResultSet(ResultSetInterface rsI) throws DEException {
        RowSetInterface roI = rsI.getRowSet();
```

```
        // loop and find row that are new but exclude those marked deleted
        RsIterator rsT = rsI.findInit(roI.ROW_New, roI.ROW_MarkDeleted, true)
        while (rsT.next()!=roI.UNDEFINED_CONTEXT) {
                (do your logic here)
        }
}
/*************************************************************************/
```

system.applicationinterface.RowSetInterface

This interface provides methods to get and put data (columns) onto a row, inspect row flags (new, updated, markdeleted, etc), add rows to result set, get original value selected for an Object Id, and so on.

Here are some commonly used methods provided by the interface (see javadoc for more explanations)

- getStringValue(String sColumnName): Return the value in the column specified with Object ID. If the value is blank or null in the database, an empty string will be returned (not a null).
- setStringValue(String value,String sObjectId): Set a value into the column specified with Object ID

Example: Getting and setting data with RowSetInterface

```
/*************************************************************************/
private void loadSettings (ResultSetInterface rsI) throws DEException {
        RowSetInterface roI = rsI.getRowSet();
        /* get a value */
        String sMethod = roi.getStringValue("S_PO_AUTO_NUM_TYPE");
        /* set a value */
        if (sMethod.trim().equals(""))
                roi.setStringValue("S","S_PO_AUTO_NUM_TYPE");
  }
}
/*************************************************************************/
```

system.applicationinterface.AppInterface

Provide methods to get various global settings independent of the result set you are in such as current App Id, System Name, Language No and User Id.

AppInterface is obtained via the ResultSetInterface's getApplication method.

Here are some commonly used methods provided by the interface (see javadoc for more explanations)

- **getSystemName()**: Return system name selected by user when login.
- **getUserId()**: Return ID of login user.
- **getConstant(String constId)**: Return the object containing the application constant identified in the ID

Example: Getting User Id with AppInterface

```
/*************************************************************************/
private String getUserId (ResultSetInterface rsI) throws DEException {
        AppInterface appI = rsI.getApplication();
        String sUserId = appI.getUserId();
        return sUserId;
}
/*************************************************************************/
```

Example: Getting the current application ID

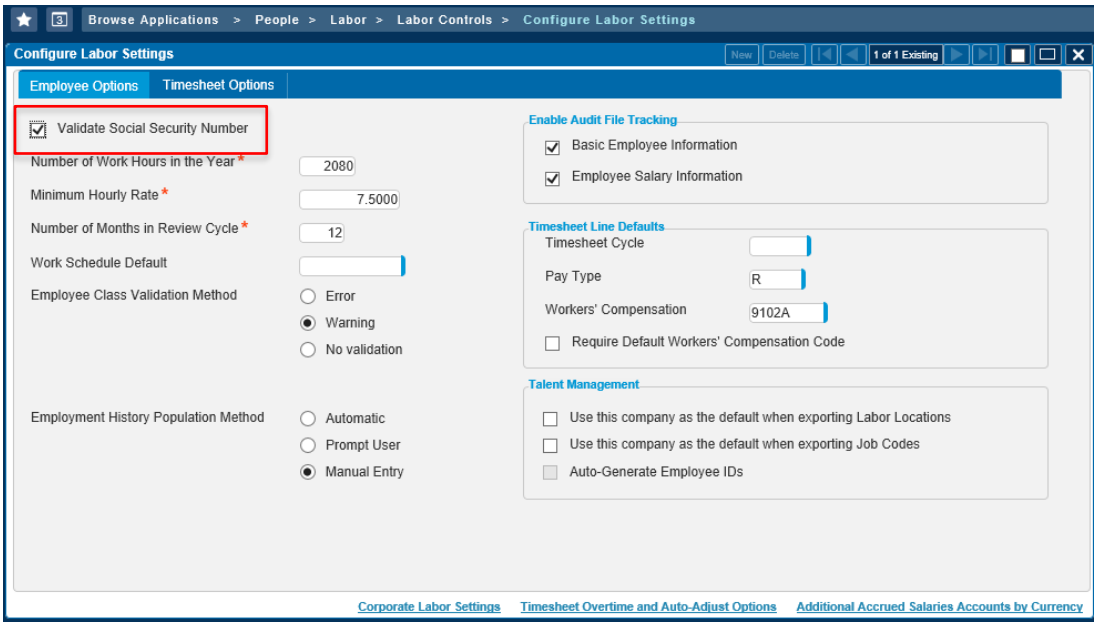
```
/*************************************************************************/
private String getAppId (ResultSetInterface rsI) throws DEException {
        AppInterface appI = rsI.getApplication();
        String appId = appI.getAppName();
        return appId;
}
/*************************************************************************/
```

Global Constants

The framework stores data that are set in setting tables (such GL Setting, PO Setting, and so on) as global constant. Data is retrieved once by the framework from the database and stored in static hashtables by framework.

Application does not have to use SQL to get the value. Application can simply get the value via the getConstant method.

Example: Getting the Validate Social Security Number Flag in Labor Settings



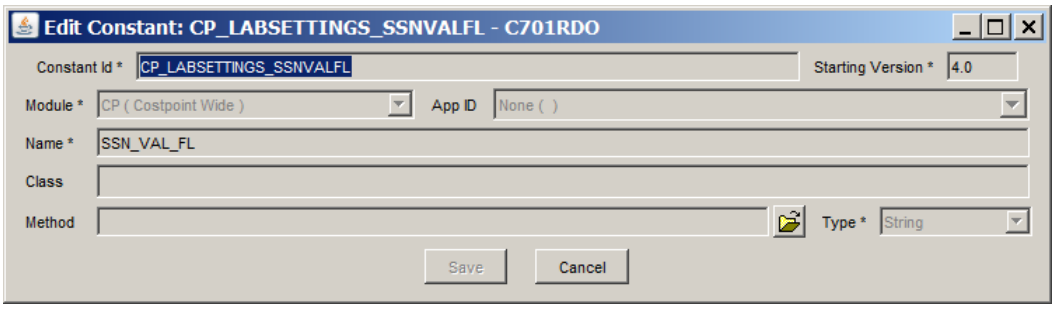
```
/*************************************************************************/
private String getLabSettingSSNFl (ResultSetInterface rsI) throws DEException {
        AppInterface appI = rsI.getApplication();
        String valSSNFl = (String) appI.getConstant("CP_LABSETTINGS_SSNVALFL");
        return valSSNFl;
}
/*************************************************************************/
```

Example - Get Project Control's top level length

```
/*************************************************************************/
private short validateSomeObjects (ResultSetInterface rsI) throws DEException {
        // get app interface from ResultsetInterface
        AppInterface api = rsI.getApplication();
        // get top level len
    Integer nTopLvlLenNo = (Integer) api.getConstant("CP_PROJCNTL_TOPLVLLENNO");
    ..........
}
/*************************************************************************/
```

List of Global Constants

In the Designer, go to Constant and filter with ID starting with CP only. Open the constant. Any constant that does not have an entry in the Class and Method fields is a global constant.



Using Constants as Variable in the Designer

To use these constants as variable in the Designer, you must assign the constant to the application in the Designer. Then you can use it as part of the formula for Editable, Visible, Value, and so on. You can also use it as part of the text for labels or status text.

Automatic Constant

Automatic constant is a subset of global constant that is always available for use. You do not have to assign them to your application. These are:

- CP_USER_ID = ID of the user who currently logs in.
- CP_LANG_NO = Language No being used by the current login session.

- CP_COMPANY_ID = Company the user is working on in the current login session.
- CP_APP_ID = App ID the user is working on in the current login session.
- CP_CURRENT_DATE = Date time of the server when user logins to an application
- CP_SESSION_ID = Station ID assigned to the user when login to Costpoint Web.
- CP_GLCONFIG_REFSTRUC1LBL = Ref1 Label
- CP_GLCONFIG_REFSTRUC2LBL = Ref2 Label
- CP_WUSERCOMPANY_SUPPRESSCSTFL = User Company Cost Suppression Flag
- CP_WUSERCOMPANY_SUPPRESSLABFL = User Company Labor Suppression Flag
- CP_WUSERCOMPANY_SUPPRESSPRCFL = User Company Price Suppresion Flag
- CP_WF_KEY
- CP_WF_CASE_KEY
- CP_WF_ACTIVITY_KEY
- CP_WF_OPTIONAL_KEY

Creating your own application constants

In addition to the global constants, you can create your own constants via your own java class. Once it is created, you can use it in your application by assigning it to your app in the Designer. It will be then available to your server code as well as client code (formula, label, status text set in Designer).

The object returned can be String object, or a Date, Integer or Double.

If you return a String containing a date, then you must format the string with the system date format (yyyy-MM-dd). Using other format will interfere with the system trying to reformat the date to the correct locale of the user. Generally, we suggest that you return a Date instead. The system will automatically convert the date to the proper format for presentation and you do not need to know how what format to use if you do it yourself. In addition, you can also use the Date object at the server for other purposes.

To create your own application constant:

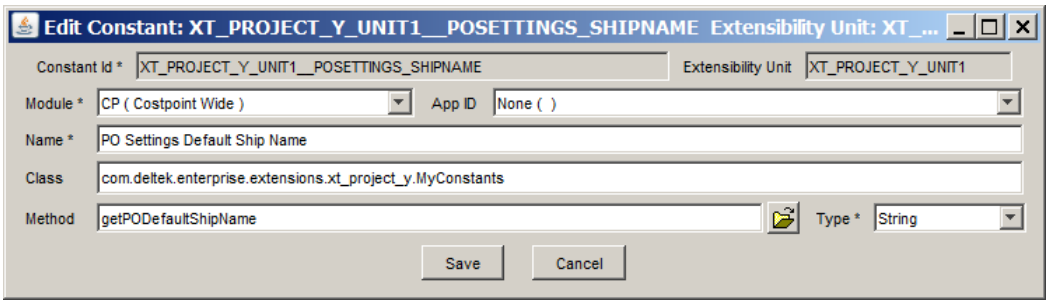
1. In the Designer: Create a new constant Id. Specify the java class and method used to populate the value of this constant. Then assign constant to your application
2. Create the java class and method to return the value. Class can be an existing class in your extensions package.
3. The java method must have this signature

   public Object xxxx (AppInterface appI) throws DEException {}
4. Use the AppInterface.getSqlManager method to get an instance of SqlManager.

Example: Create constant Ship Name from the Default Ship ID in the PO_SETTINGS

/**/

Designer:



Java class:

```
/************************************************************************/
package com.deltek.enterprise.extensions.xt_project_y;
import java.sql.*;
import com.deltek.enterprise.system.applicationinterface.*;

public class MyConstants {
    public String defaultShipId;
    public String defaultShipName;
    public Object getPODefaultShipName (AppInterface appI) throws DEException, SQLException {

            SqlManager sqlMgr = appI.getSqlManager("DATA",this);
            String sysName = appI. getSystemName();
            /* Get global constant Default Ship ID */
            defaultShipId = (String) appI.getConstant("CP_POSETTINGS_DFLTSHIPID");
            /* Select ship name from ship id */
            String Select = "SELECT SHIP_NAME FROM SHIP_ID “ +
                        “WHERE SHIP_ID = :defaultShipId INTO defaultShipName";
            sqlMgr.SqlExecuteQuery(Select);
            return defaultShipName;
    }
}
/************************************************************************/
```

system.applicationinterface.LoggerInterface

All logging in Java code should be done using the provided LoggerInterface. Avoid using System.out. Use LoggerInterface to enable more flexibility and extensibility in logging information. The advantages of using the Logger are:

- It is possible to print to the console or some log files by changing properties in the config file.
- No need to comment/uncomment System.out.println in the code

- The logging messages from a particular package or a particular dir can be filtered for analysis/debugging purpose

How to Add Logging in Your Class:

- Import com.deltek.enterprise.system.applicationinterface.LoggerInterface
- Get a LoggerInterface from AppInterface
- Use the logging methods from the LoggerInterface.

Example: Using the LoggerInterface

```
/*********************************************************************/
import com.deltek.enterprise.system.applicationinterface.LoggerInterface;
import com.deltek.enterprise.system.applicationinterface.DEException;
import com.deltek.enterprise.system.applicationinterface.ResultSetInterface;
import com.deltek.enterprise.system.applicationinterface.AppInterface;

public class myClass {
    public short validateMyEntity (ResultSetInterface rsI) throws DEException {
        AppInterface appI = rsI.getApplication();
        LoggerInterface logger = appI.getAppLogger();

        logger.debug("This a debug message");
        logger.info("This is a info message");
        logger.warn("This is a warning message");
        logger.error("This is an error message");
        logger.fatal("This is a fatal message");
    }
}
/*********************************************************************/
```

CPlog4j.properties

At run time, the system will read the default configuration file CPlog4j.properties located in the classpath (currently located at \applications\enterprise\properties). This file contains the specified output, severity level and layout of messages.

Severity levels are (from lowest to highest): Debug, Info, Warn, Error, Fatal. The severity level set in the configuration file indicates the lowest to be output. For example, if the level is set at Warn, then only Warn, Error and Fatal level messages are output.

system.applicationinterface.CPConstants

The CPConstants interface contains constants that are used system wide. These are java constants. Do not confuse this with Costpoint global constants that were discussed in the AppInterface section.

For example, CP_OK, which corresponds to 0, can be returned from the validation when no errors are encountered. Other constants in this interface include CP_ACCT_DELIM, CP_MAX_PROJ_LVLS, and CP_MAX_NUM_SUB_PDS.

To use CPConstants, import com.deltek.enterprise.system.applicationinterface.CPConstants. If your class implements CPConstants, you can use the constants such as CP_OK without the qualifier. Otherwise, you need to use them with the interface CPConstants as the qualifier, for example, CPConstants.CP_OK

Example: Class using CPConstants

```
/*************************************************************************/
import com.deltek.enterprise. system.applicationinterface.CPConstants;
public class PjmbasicMyClass {
        public short retcode = CPConstants.CP_OK;
        ...........
}
/*************************************************************************/
```

system.applicationinterface.SqlManager

SqlManager class is used to perform SQL operation on the database. You request an instance of SqlManager from ResultSetInterface (or AppInterface or ActionInterface). With it, you can use it's method to execute SQL statement.

Rules

- Avoid excessive request of SqlManager instances. For example, do not request SqlManager in a loop. Request one instance and then use it in the loop. When writing common validation method, use a passed in SqlManager as a parameter instead of creating a new SqlManager.

  The system will throw an application error if more than 20 SqlManager instances are requested within one single plug-in call (an application method called by the system for a single event). This includes all SqlManagers requested from functions invoked from the plug-in class.

- Never add table owner in front of table name. Owner is assumed by the name setup in the server connection pool.
- DbF functions are provided to cover cross DBMS SQL syntax. Use them in SQL statement like a SQL function (see an example below).

Some commonly used methods in SqlManager are:

- **SqlPrepareAndExecute(String)**: prepares and executes SQL statement.
- **boolean SqlFetchNext()**: fetches next row from result set.
- **SqlExecSP(sExpectedVersion,bCheckOnly,sSPName,sParams,bCommit)**: invokes stored procedure.
- **SqlGetModifiedRows**: Get the number of rows affected by the last SQL statement.
- **boolean SqlExecuteQuery(String)**: executes select statement and fetchs first row.

Example: SqlExecuteQuery using class variables for bind and into variables

```
/*************************************************************************/
```

```
/* Bind variables are declared as class variables. SqlManager uses reflection to read and
    write values back to your object. All bind variables are preceded with the colon (:) in
    the SQL statement.

    If your class has been extended and the SQL is being done on the subclass, do not use
    class variables as bind variables. Use the local hashmap approach (see next example)
*/


public String sShipId;
public String sShipName;


public void posettingsGetShipDesc (ResultSetInterface rsI) throws SQLException, DEException {
    SqlManager sqm = rsI.getSqlManager(this);
    RowSetInterface roI = rsI.getRowSet();
    sShipId = roI.getStringValue("PO_SETTINGS___DFLT_SHIP_ID");


    String sSelect = new StringBuffer()
             .append("SELECT SHIP_NAME FROM SHIP_ID WHERE SHIP_ID = :sShipId")
             .append(" INTO :sShipName").toString();
    sqm.SqlExecuteQuery(sSelect);
    roI.setStringValue(sShipName,"SHIP_NAME");
}
/*********************************************************************/
```

Example: SqlExecuteQuery using a local hashmap to hold bind/ into variables

```
/*********************************************************************/
public void posettingsGetShipDesc (ResultSetInterface rsI) throws SQLException, DEException {
    SqlManager sqm = rsI.getSqlManager(this);
    RowSetInterface roI = rsI.getRowSet();
    String sShipId = roI.getStringValue("PO_SETTINGS___DFLT_SHIP_ID");
    String sShipName = null;
    // put variables into a local HashMap
    HashMap bindHash = new HashMap();
    bindHash.put("sShipId ", sShipId);
    bindHash.put("sShipName ", sShipName);
    // Put HashMap in SqlManager
    sqlMgr.setBindClasses(null,null,bindHash);


    String sSelect = new StringBuffer()
             .append("SELECT SHIP_NAME FROM SHIP_ID WHERE SHIP_ID = :sShipId")
             .append(" INTO :sShipName").toString();
```

```
    sqm.SqlExecuteQuery(sSelect);
    // Get the value out of hashmap
    HashMap outHash = sqlMgr.getBindsTable();
    sShipName = (String) outHash.get("sShipName");
    roI.setStringValue(sShipName,"SHIP_NAME");
}
/*********************************************************************/
```

Using DbF functions

If your customer runs Costpoint on multiple DMBS platform, your SQL statement must work for all platform. DbF functions are provided to avoid different syntax for different DBMS platform. For example, outer join syntax is different for SQL Server versus Oracle. Costpoint provides the DbF_OuterJoin function to avoid having to write different SQL. Alternatively, you can write SQL in ANSI syntax which would work for all platform.

DbF function is used like a SQL function (not a java function). Therefore, do not concatenate your SQL statement with DbF function java calls. Include such function inside the SQL statement. The system will parse the DbF function inside the statement to the appropriate back end syntax.

For a complete list of DbF functions, open the Sql Editor in the Designer and select the drop down box for DbF functions.

Example: Using DbF_Outerjoin function

DbF_Outerjoin automatically converts to ANSI outerjoin syntax after the system parses the statement.

```
/*********************************************************************/
String select  = "SELECT T1.USER_ID, T1.EMPL_ID, T2.LAST_NAME FROM USER_ID T1, EMPL T2 WHERE
DbF_OuterJoin(T1.EMPL_ID,T2.EMPL_ID)"
/*********************************************************************/
```

Example: Using DbF_OptionalString to insert NULL when field is empty

SqlManager will insert a blank string if the value passed in is empty. Use DbF_OptionalString for column that needs to be NULL if the value is blank.

```
/*********************************************************************/
String sInsert = "INSERT INTO PO_HDR_DFLT (PO_ID, PO_RLSE_NO,RQ_ID,SHIP_ID,……) " +
" VALUES (:sPoId, :nPoRlseNo, DbF_OptionalString(:DfltShipId), …..)
sqlMgr.SqlPrepareAndExecute(sInsert);
/*********************************************************************/
```

Using other key words

- Key word CP_USER_ID can be used to substitute the value of the User Id in SQL statement. Use it for MODIFIED_BY column (see example below)
- Key word CP_COMPANY_ID can be used to substitute the value of the current company id the user is working with in the current session.
- Key word CP_APP_ID can be used to substitute the value of the current app id the user is working with in the current session.

- Key word CP_CURRENT_DATE can be used to substitute the value of the current database server date and time. Alternatively, you can also use the DbF_DateCurrent function.

Example: Using key word: CP_USER_ID and DbF functions

```
/*********************************************************************/
sProjId = rowsetI.getStringValue("PROJ_ID");
sOldOrgId = rowsetI.getStringValue("ORG_HIST___OLD_ORG_ID");
String sSql = new StringBuffer()
        .append("UPDATE PROJ SET ORG_ID = :sOldOrgId,”)
        .append(“MODIFIED_BY = :CP_USER_ID,TIME_STAMP = DbF_DateCurrent()”)
        .append(“ WHERE PROJ_ID LIKE :sProjId DbF_ConcatChar() '%'").toString();
sqlMgr.SqlPrepareAndExecute(sSql);
/*********************************************************************/
```

Example: Checking for concurrency

Concurrency error should be captured as a FATAL error.

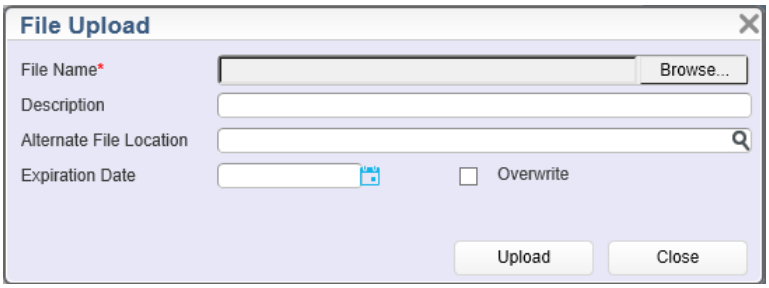
```
/*********************************************************************/
sqlMgr.SqlExecute();
if (sqlMgr.SqlGetModifiedRows() == 0)
{ // NO row updated by last UPDATE statement
        String tableName = “xxxxx”;
        String dbOperation = “xxxx”
   String[] msg = { tableName, dbOperation }; //table name, operation
   rsI.addRSMessage("XT_PROJECT_Y_NO_ROWS_UPDATED",rsI.INFO, msg);
   return rsI.INFO;
}
/*********************************************************************/
```

system.applicationinterface.FileHandlerInterface

File Uploading

For files that must be processed by Costpoint (for example, preprocessor input files), the user can upload the file using the File Upload Manager via the Process->File Upload option on the menu. This uploads the file to the Costpoint database table W_USER_FILE_CATLG. Files can also be placed in an alternate file location (see next section).

File Uploading UI



A dialog appears when clicking on the File Upload option on the menu bar.

- Enter the file name or Browse to the file (located in your drive or a network drive).
- Enter a brief description.
- Enter the expiration date. A purge process can be scheduled to remove these files from the database based on the expiration date.
- Check Overwrite box if you want to overwrite a previous version. (Otherwise, an error message may be returned if a duplicate file is found).

The file will be saved in the table W_USER_FILE_CATLG with the same file name. The following are stored as key for this table: User ID, Company ID, App ID and file name

Alternate location

To process a file, the file must either be uploaded in the database or uploaded in an alternate location (disk file).

An alternate location is a folder where the application server can access/read/write disk files. Once the folders are created on disk, you associate the URL with a logical name inside Costpoint. An alternate file location is specified in the Manage Alternate File Locations application. After a file is placed in that location, it can be used in the application if the application screen provides input fields for entering file location ID and file name.

FileHandlerInterface

To access files already uploaded for processing, use the FileHandlerInterface in com.deltek.enterprise.system.applicationinterface package.

Common methods in FileHandlerInterface include:

- boolean **fileOpen**(java.lang.String fileName, int nStyle): Open a DB file uploaded in database
- boolean **fileOpen**(java.lang.String fileName, int nStyle, String altLocation): Open a file uploaded in alternate location
- String **fileRead**(int len): Read a buffer of characters from a file to a string starting with the current position to the new position (current plus len passed in).
- String getGetStr():Read next line as String from an opened file.
- void **fileClose** (): Close an opened file.

Example: Coying a file uploaded in DB to a new file in DB

/**/

```
import com.deltek.enterprise.system.applicationinterface.FileHandlerInterface;
public class TFileMgr {
    public short copyFile (ActionInterface aci) throws DEException {
        FileHandlerInterface fi = aci.getFileManager();
        ResultSetInterface rsi = aci.getResultSet();
        RowSetInterface roi =  rsi.getRowSet();
        String fileName = roi.getStringValue("MY_RS_FILE_NAME");
        fi.fileCopy(fileName,fileName+"_copy",true);
        return 0;
    }
}
/*********************************************************************/
```

Example: Reading a file uploaded in DB

```
/*********************************************************************/
import com.deltek.enterprise.system.applicationinterface.FileHandlerInterface;
public class TFileMgr {
    public short copyFile (ActionInterface aci) throws DEException {
        FileHandlerInterface fi = aci.getFileManager();
        ResultSetInterface rsi = aci.getResultSet();
        RowSetInterface roi =  rsi.getRowSet();
        String fileName = roi.getStringValue("MY_RS_FILE_NAME");
        if (!fi.fileOpen(fileName,fi.OF_Read)) {
              rsi.addMessage(....);
              return rsi.ERROR;
        }
        String tempStr = null;
       /* read one line at a time */
        while ((tempStr = fi.fileGetStr()) != null) {
            ... process data in tempStr...

        }
        fi.fileClose();
        return 0;
    }
}
/*********************************************************************/
```

Example: Copying a file in db and write out to a disk file in alternate location

```
/*********************************************************************/
```

```
public void copyFileToAltLocation(ActionInterface action) throws DEException {
    RowSetInterface row = action.getResultSet().getRowSet();
    String altLocation = row.getStringValue("ALT_LOCATION");
    String fileName = row.getStringValue("FILE_NAME");
    String newFileName = row.getStringValue("ALT_FILE_NAME");

    // open file
    FileHandlerInterface fileHandler = action.getFileManager();
    if (fileHandler.fileOpen(fileName, fileHandler.OF_Read)) {
      String content = fileHandler.fileRead(fileHandler.length());
      fileHandler.fileClose();
      // create new file
      newFileName = (newFileName == null) ? fileName : newFileName;
      if (fileHandler.fileOpen(newFileName, fileHandler.OF_Create, altLocation)) {
        fileHandler.fileWrite(content);
        fileHandler.fileClose();
        // Ok
        String[] s = {newFileName};
        action.addMessage("XT_PROJECT_Y_FILE_CREATE_OK", action.INFORMATION, s);
      }
      else {
        // cannot create new file
        String[] s = {newFileName};
        action.addMessage("XT_PROJECT_Y_FILE_CREATE_FAILED", action.ERROR, s);
      }
    }
    else {
      // existing file is not found
      String[] s = {fileName};
      action.addMessage("XT_PROJECT_Y_FILE_NOT_FOUND", action.ERROR, s);
    }
}
/*********************************************************************/
```

Chapter 3: system.utils Package

In addition to system's applicateinterface package, application can also import and use classes in the system's utils package.

system.utils.Numbers

Provides methods to do rounding, compare, truncate, and so on.

Here are some commonly used methods provided by the class.

- public static int compare(double number1, double number2): returns 1 if number1 > number2, 0 if number1=number2, else return -1

  This method will help avoid errors caused by floating point arithmetic and will allow reliable comparison of two double values

  For example, the number in regular, decimal, format 123.45 is stored as 1.2345E02; one is stored as 1.0E01. This means that java uses quite different rules (floating point arithmetic) to do any arithmetic operation. Rounding errors can happen when you use two or more double variables or just add some number to the double variable.

  For example:

  - double d1=1.1233357
  - double d2=4.165
  - double d=d1-d2
  - double d3=d1-d2-d

  We would expect d3 to equal zero. However, d3 is actually = 2.220446049250313E-16.

  Use this method compare instead of doing your own arithmetic for comparison.
- public static double round(double value, int scale)

  Round a number to the scale specified.

System.utils.UIFormat

Provide methods to format number or date to string for presentation purposes.

Here are some commonly used methods provided by the class.

- public static String numberToStr(AppInterface appI, double amount, int nScale, boolean bThousSep,boolean bPadDec)
- public static String dateTimeToStr(AppInterface appI,Calendar datetime, boolean bDate,boolean bTime, boolean bFourDigitYear)

Example: Formatting a currency amount in an error message

```
/*****************************************************************/
import com.deltek.enterprise.system.applicationinterface.*;
import com.deltek.enterprise.system.utils.UIFormat;


public class MyRowValidation {
```

```
    public short validateRow (ResultSetInterface rsi) throws DEException {
        RowSetInterface roi = rsi.getRowSet();
        double parentAmt = roi.getdouble("RQ_LN_TOT_CALC_AMT");
        double childTotAmt = roi.getdouble("RQ_LN_TOT_AMT");

        if (parentChangeAmt != childChangeAmt) {
            AppInterface app = rsi.getApplication();
            String parAmt = UIFormat.currencyToStr(app, parentAmt,true,true,true);
            String childAmt = UIFormat.currencyToStr(app, childAmt,true,true,true);
            rsi.addObjectMessage("TOT_AMT_OBJ_ID","XT_PROJECT_Y_AMT_INVALID",rsi.ERROR,
                new String[] {parAmt,childAmt});
            return rsi.ERROR;
        }
        return 0;
    }
/*********************************************************************/
```

Chapter 4: Plug In Events

Plug-in Classes Are Stateless

Application plug-in classes invoked by the framework (such as object/cell/line validations, actions, etc) are stateless objects. Between method calls, class variables can be of any state since the framework does not clear its cache. Therefore, class variables should be initialized inside the method when the method is called as a plug in entrance.

For example: You have two object validation methods implemented in the same class. These two methods will be invoked independently. Do not set class variables in one method that will be used for the other method.

Note: If one method in the plug-in class is invoking other private method in the same class, no re-initialization is necessary in the other private methods since this is still within the plug-in lifetime. Initialization should happen only at the beginning of those public methods, which are called by the framework.

RS Population Event

RS Populate Event

- RS Populate event happens when the framework executes the data retrieval for a result set. This can be initiated by user selects Query on the result set or the result set auto populates its data on opening.
- On this event, the framework will select the data based on the SQL statement entered in the Designer for result set. If you register a class for the RS Populate event, the class will be called by the framework before the data is sent to the client.
- RS Populate class is used to manipulate the data beyond the Select statement. For example, getting additional data (such as description or name from another table) where additional table join in the select SQL is costly for performance or not possible. It is also used to populate or add rows to the result set that are not coming from the database.

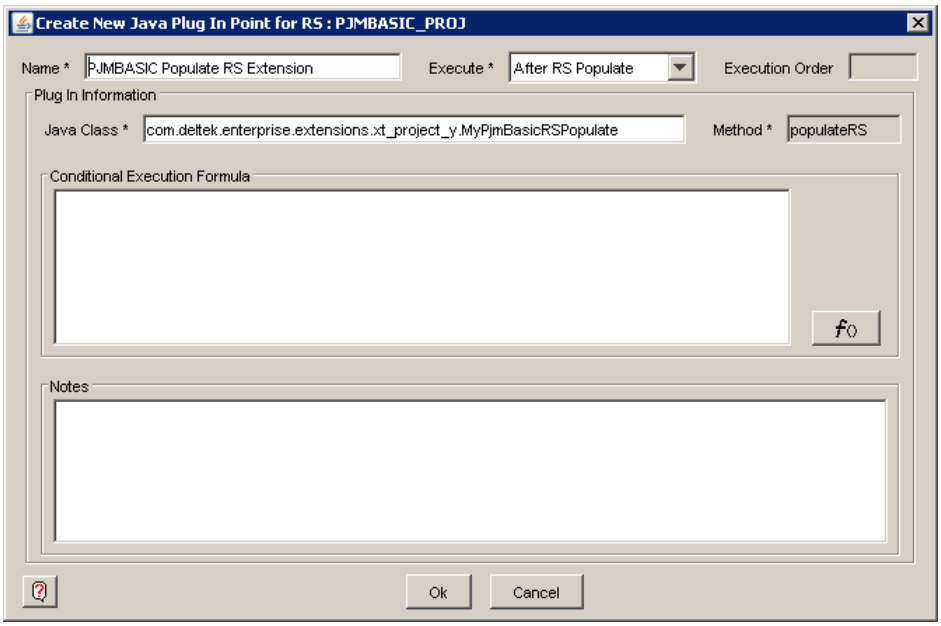
Caching Scheme

The framework is optimized to retrieve the rows from the database only as needed. It uses cache and buffering mechanism to retrieve only enough data to fill the UI screen. Each time the system needs to fetch additional rows (on user scrolling at the client), it will go to the database and fetch more. The size is indicated in DB_FETCH_SIZE setting in enterprise.properties. As user scrolls for more data, the server will check its cache before going to the database to get more.

The cache size is indicated by the RS_CACHE_SIZE setting. Once the rows reached the RS_CACHE_SIZE, old rows will be discarded to make room for new rows. New rows entered by user or rows that have been edited are stored separately and not included in this cache size limit.

Due to this scheme, RS populate plug-in class may be called more than one time. So do not use class variables in the method unless they are initialized each time.

Extensibility



- Enter a descriptive name for the plug-in. Select After RS Populate. Your class will be executed after the standard Deltek RS Populate plug-in.
- Enter the fully qualified package and class name. Your java class must implement PopulateRSInterface interface and the method populateRS.
- Use method nextBufferRow() from the ResultSetInterface to loop through rows retrieved by the framework. The system sets context automaticaly so you do not need to call method setRowSetContext in the loop.

Example: Looping through rows selected by framework and set field values

```
/*****************************************************************************/
package com.deltek.enterprise.extensions.xt_project_y;
import com.deltek.enterprise.system.applicationinterface.*;

public class MyPjmBasicRSPopulate implements PopulateRSInterface {
   public void populateRS(ResultSetInterface rsI) throws DEException {
      RowSetInterface rowSetI = rsI.getRowSet();
      int i;
      while ( (i= rsI.nextBufferRow())!=rowSetI.UNDEFINED_CONTEXT){ {
          rowSetI.setStringValue("","CURRENT_FISCALYEAR");
          if (rowSetI.getStringValue("REV_LBL").equals("N/A")) {
             rowSetI.setStringValue(null,"REV_LBL");
             rowSetI.setDouble(null,"REV_AMT");
          }
      }
```

```
    }
}
/*************************************************************************/
```

Validation event

Types of Validation Events

Validations are plug-in events fired by the framework so the application will have a chance to validate the data.

There are three types of validations: Object validation, Line (or Row) Validation, and RS Validation. The timing of the events at the server depends on the Validate Frequency mode on login.

Rowset definition

A rowset is a group of rows within a result set having the same parent row. Thus, top-level result set in an application will always contain only one rowset. A non top-level result set can have multiple rowset if there are multiple parent rows.

For example:

Two voucher headers A and B each having two voucher lines: A1, A2 and B1, B2 respectively. In such example, there are two result sets: Voucher Header and Voucher Line. Since Voucher Header is a top-level result set, there is only one rowset with two rows A and B. For the Voucher Line result set, there are two rowsets: A1 and A2 with parent A, and B1 & B2 with parent B.

This explanation will help you understand the discussion in the rest of this chapter.

Object Validation

Object Validation should be used to validate a single object.

Timing

In Field mode, Object Validation is invoked when value in the object has changed and user tabs out of it. Once validation is done, it is not called again when user just tabs through it again unless the value is changed again. If the field's value is changed programmatically by the application java code, no Object Validation is fired.

In Line mode, Object Validation is delayed and called when user leaves the current line to another line and the object value has been changed. So while user keep changing the value but still stay on the same line, it is not yet called.

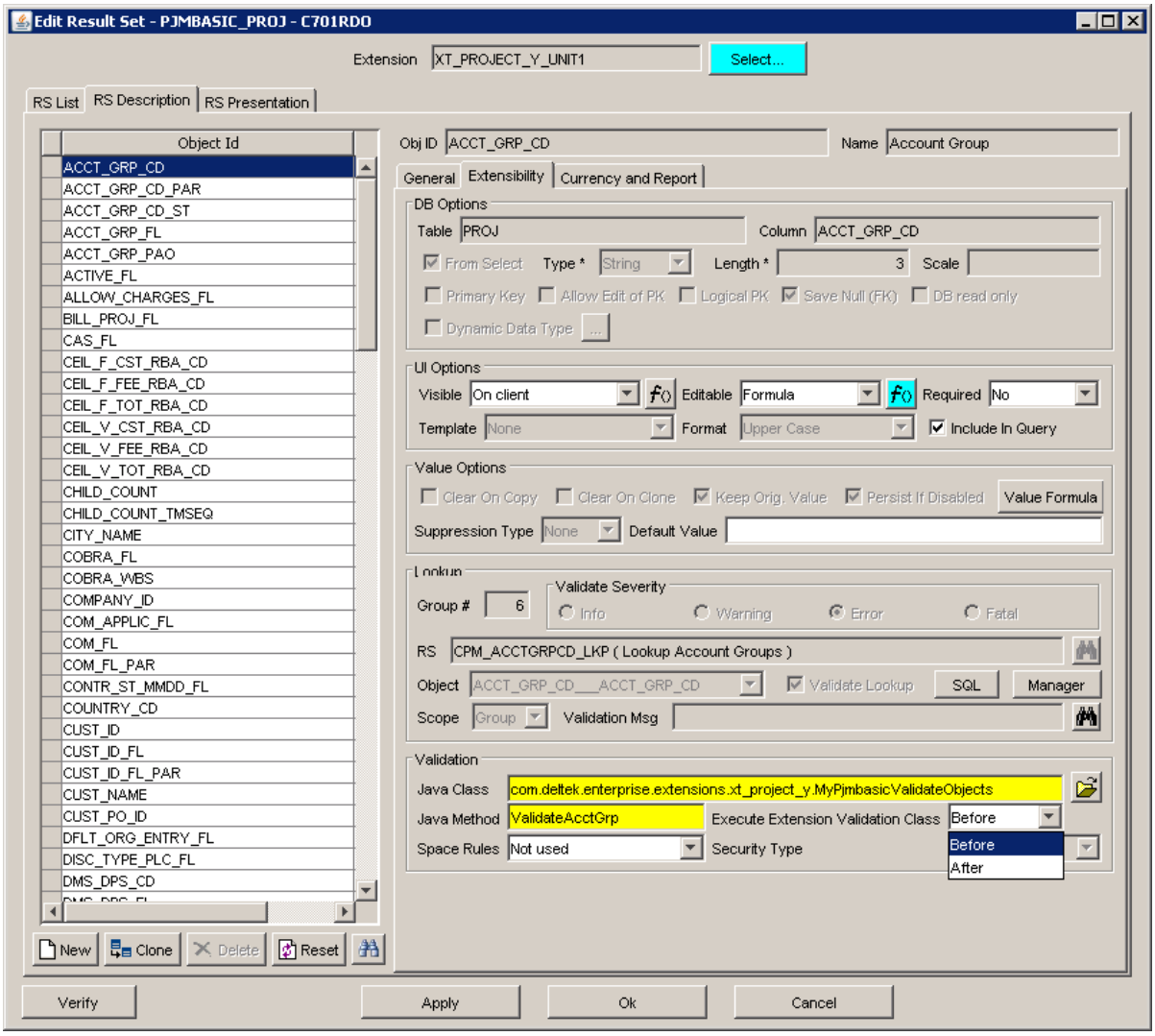
In Application mode or web service mode, Object Validation is only called on Save.

Rules

- Object validation is never invoked if the value is changed to a blank or null. It is only invoked if the value has been changed to a non-blank value. Therefore, you should never have Object Validation code that checks if the field is empty because the plug-in code would never be executed. If you need to check that the field is empty to do something, do this check in the Row Validation plug-in class.

- Object validation should be written to look at the value of a single object only. If the validation depends on the value of other objects on the screen, it must be done in Row validation or RS validation when user has completed the input for the dependent fields.

Extensibility



- When extending existing Deltek standard Object Validation, you can create the plug-in class to be executed before or after the standard Deltek Object Validation is fired.
- Enter the full package name of the plug-in class. Enter the method name.
- Select Before or After execution of the standard Deltek Object Validation.
- Implement the class and the method.

Example: Object validation

```
/**************************************************************************/
package com.deltek.enterprise.extensions.xt_project_y;
import com.deltek.enterprise.system.applicationinterface.*;
```

```
public class MyPjmbasicValidateObjects {
    public short ValidateAcctGrp(ResultSetInterface rsI) throws SQLException, DEException {
        sqlMgr = rsI.getSqlManager(this);
        RowSetInterface rowsetI = rsI.getRowSet();
     /* In object validation, rsi.getObjectId() method
   returns the object Id of the object being validated */
        String sAcctGrpObjId = rsI.getObjectId();
        String sAcctGrpCd = rowsetI.getStringValue(sAcctGrpObjId);
        short nReturn = CP_OK;
        String sSql = new StringBuffer()
            .append("SELECT ACCT_GRP_CD FROM ACCT_GRP_CD ")
            .append("WHERE ACCT_GRP_CD = :sAcctGrpCd ")
            .toString();
        if (!(sqlMgr.SqlExecuteQuery(sSql))) {
            String sMsgId = "CP_ACCTGRP_INVALID";
            rsI.addObjectMessage(sAcctGrpObjId, sMsgId ,rsI.ERROR);
            nReturn = rsI.ERROR;
        }
        return nReturn;
    }
}
/*************************************************************************/
```

Row Validation

Row Validation should be used to validate each row found in a result set (RS).

Timing

In Field mode and Line mode, Row Validation is invoked when at least one field on the row has changed and user moves the focus to another row.

In Application mode or web service mode, Row Validation is delayed and is only called on Save.

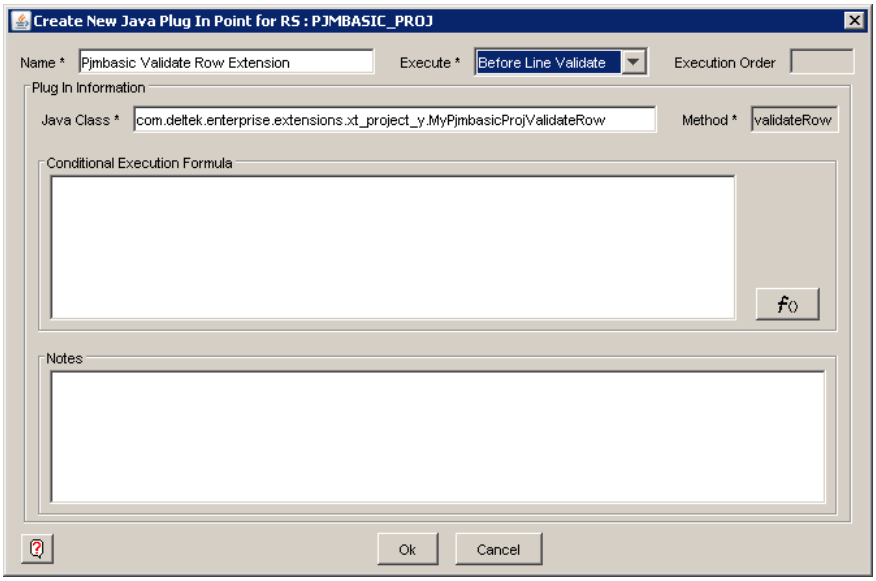
Rules

- Row Validation is never called when the row is marked deleted. Therefore, you should never have Row Validation code that checks if the line is marked deleted because the code would never be executed. If you need to check this, do this check in the RS Validation plug-in class.
- Row validation should be written to validate objects' relationship within a line. If the validation logic depends on the value of other objects on other lines (like getting a total, checking for duplicate value, and so on), you should put this logic in RS validation.

- If you have parent and child RS relationship and parent ID or key columns needs to be cascaded/copied down to the child RS on insert of new rows, you need to do this in places other than in the row validation of the child RS.

  The reason is in Field or Record mode, row validation of the child RS may occur and then user decides to change the parent column value (for example, PROJ_ID, VEND_ID, etc,). Since child row is already validated, it will not get validated again when the user changes the value in the parent RS

Extensibility



- When extending existing Deltek standard Row Validation, you can create the plug-in class to be executed before or after the standard Deltek Row Validation is fired. If you need to do both, create two plug-in classes, one for each.
- Enter a descriptive name for the plug-in. Select Before Line Validate or After Line Validate.
- Enter the full package and name of the plug-in class. Note that the method name is always “validateRS”.
- The class must implements RSValidation interface with the method validateRS.

Example: Row Validation

```
/*****************************************************************************/
package com.deltek.enterprise.extensions.xt_project_y;
import com.deltek.enterprise.system.applicationinterface.*;

public class MyPjmbasicProjValidateRow implements RowValidation {
    public short validateRow(ResultSetInterface rsI) throws SQLException, DEException {
      RowSetInterface rowSetI = rsI.getRowSet();
      String sProjId = rowSetI.getStringValue("PROJ_ID");
      String sOrgId = rowSetI.getStringValue("ORG_ID");
      String sAcctId = rowSetI.getStringValue("ACCT_ID");
```

```
        return validatePOA(sProjId,sOrgId,sAcctId,rsI);
    }
}
/*************************************************************************/
```

RS Validation

RS Validation should be used to validate across the rows within a RS or across multiple RS.

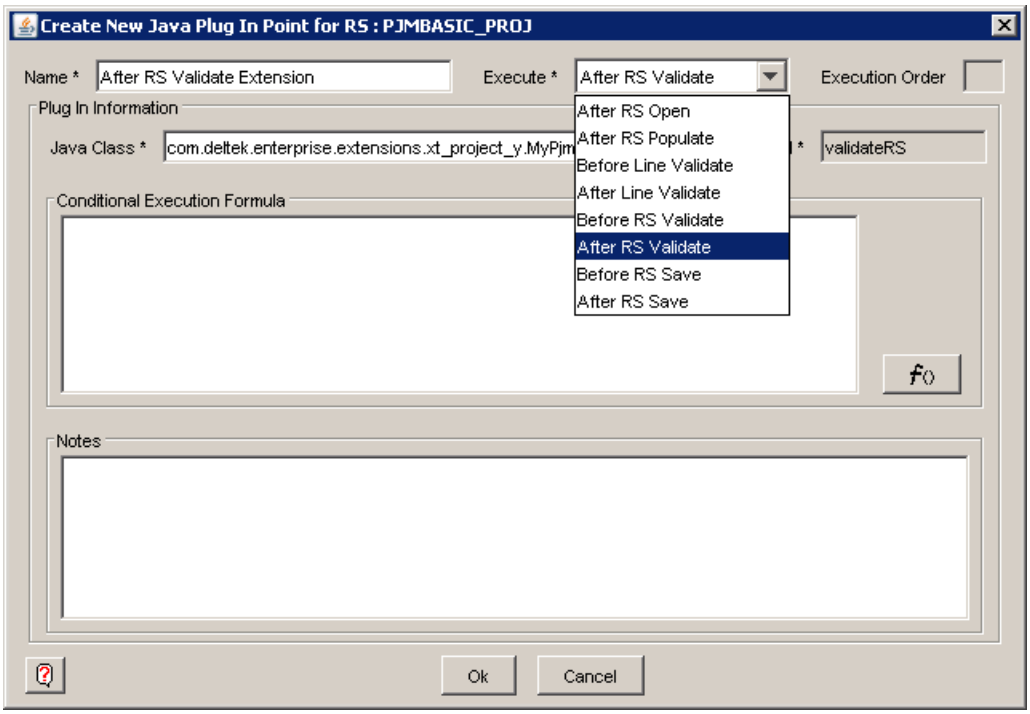
Timing

In all mode (Field, Row, Application or web service) RS Validation is called only on the Save event.

Rules

- RS Validation is not called if there is no change to any row on that result set.
- RS Validation should be written to validate objects relationship across lines. For example, when totaling amounts or percents across the lines or when checking for unique value of certain column.
- You must always iterate through the RS since the event is fired once for each RS, not each row.

Extensibility



- When extending existing Deltek standard RS validation, you can create the plug-in class to be executed before or after the standard Deltek RS validation is fired. If you need to do both, create two plug-in classes, one for each.
- Enter a descriptive name for the plug-in. Select Before RS Validate or After RS Validate.

- Enter the full package name of the plug-in class. Note that the method name is always “validateRS”.
- The class must implements RSValidation interface with the method validateRS.

Example: RS Validation

```
/*************************************************************************/
package com.deltek.enterprise.extensions.xt_project_y;
import com.deltek.enterprise.system.applicationinterface.*;

public class MyPjmbasicValidateRS implements RSValidation {
    public short validateRS(ResultSetInterface rsI) throws DEException {
      short nReturn = CP_OK;
      RowSetInterface rowSetI = rsI.getRowSet();
      /* Looping is necessary in RS Validate */
      RSIterator rsT = rsI.findInit(rowSetI.ROW_New,rowSetI.ROW_MarkDeleted,true);
      int nNextLvlNo = 1;
      while (rsT.next() != rowSetI.UNDEFINED_CONTEXT) {
         rowSetI = rsI.getRowSet();
         int nLvlNo = rowSetI.getInt("PROJ_LVL___LVL_NO");
         if (nLvlNo != nNextLvlNo) {
            rsI.addLineMessage("PJMBASIC_LVL_NUMBERING", rsI.ERROR);
            nReturn = rsI.ERROR;
         }
         nNextLvlNo += 1;
      }
      return nReturn;
   }
}
/*************************************************************************/
```

Connection Mode Summary

In Field mode:

- Object and Row validation occur during data entry when user leaves the field or the line.
- RS validation occurs when you save a record.

In Record mode:

- Row validation occurs during data entry when user leaves the line. Catch up Object validation for all objects occur right before row validations occur.
- RS Validation occurs when you save a record.

In Application mode and web service mode:

- No validation occurs during data entry. Same for web service as the whole data stream is submitted all at once.
- All validations are done when the record is saved.
- Catch up Row Validation for all rows occur right before RS validation occurs. Within Row Validation, catch up Object Validation for all objects occur before row validation occurs.

Frequency Mode	Instant Field Validation	Instant Row Validation
Field Mode	**Y**	**Y**
Record Mode	N	**Y**
Application Mode	N	N
Web Service Mode	N	N

Validations Handled by System

There are validations that are automatically handled by the system that you should be aware of before you start coding your validation:

- System will check for nulls in required columns.
- Validates the max length for each data field submitted and that the data matches the field type (for example, string, number, date).
- Validate for entries in a combo box (in case the data is being sent from a non browser client)
- If validated lookups were defined for the result set, then the system does the lookup validations before invoking app-specific validations.
- Other system validation built in to templates (such as X >0 template, and so on)

Before & After Save

During the save transaction, there are three basic events: BeforeSave, Standard Save and AfterSave.

Standard Save is done by the framework on ‘saveable’ tables represented in the RS (saveable means RS is checked for standard save and PK is present for the table). These are set in the Designer for the result set. There is no plug in for Standard Save. It is done entirely by the framework.

If you need to update other tables not represented in the RS, you can utilize the “beforeRSSave” or the “afterRSSave” event. For example, update inventory after a purchase order is saved on the Manage Purchase Order application.

If there is more than one result set in an application, beforeRSSave are issued first to the top result set and then down until the last result set at the bottom of the tree.

After BeforeRSSave for all result set are completed, Standard Save of the result set(s) will take place. Again, if there are more than one result set in an application/tree, Standard Save are done first to the top result set and then down until the last result set at the bottom of the tree. Within the standard Save for a single result set, Delete happens first, then Update and then Insert.

After standard Save for all result set are completed, AfterRSSave will take place. Again, if there are more than one result set in an application, afterRSSave are done first to the top result set and then down until the last result set at the bottom of the tree.

BeforeRSSave

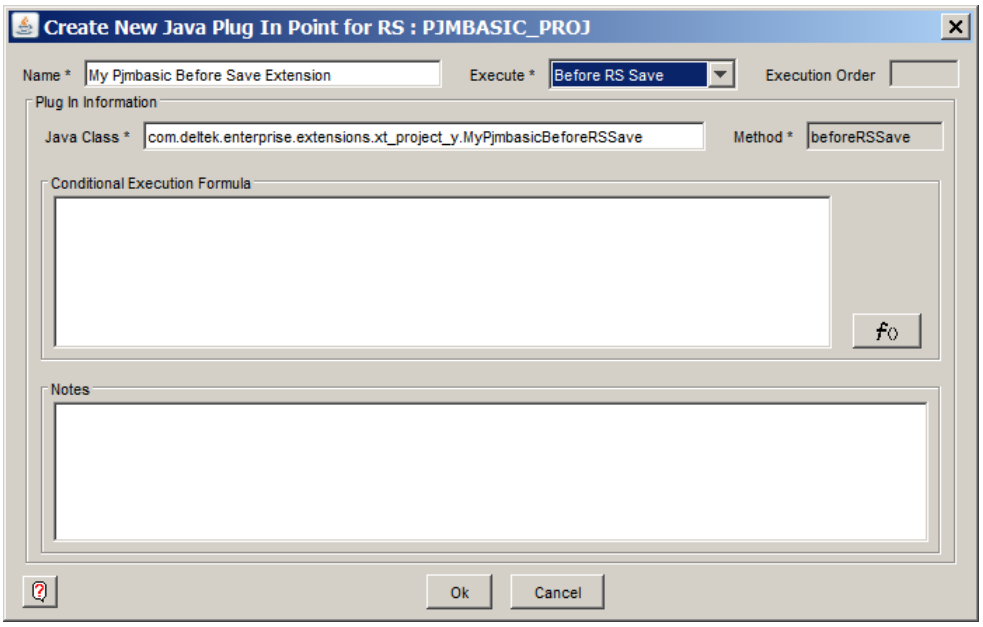
Timing

- BeforeRSSave is invoked only during a save transaction. It is never invoked during data entry regardless of connection mode.
- After all the validations are done, the system will start the database transaction. Then it calls the BeforeRSSave plug-in class.

Rules

- BeforeSave is used to save data to tables other than the main table (if these tables are needed before the main table can be saved).
- There should be no commit in this method since transaction has begun.
- BeforeRSSave can also be used to assign parent keys/ID to the children RS if the ID are not assigned in validation for fear of having gaps in ID when validation fails. If BeforeRSSave fails for some reason, all update in it will be rolled back.
- Although not recommended, you can perform validation in this plug-in and if error severity message is added to the message container (via method addObjectMessage or addRowMessage, and so on) the framework will rollback all updates done so far in the save event.

Extensibility



- Enter a descriptive name for the plug-in. Select Before RS Save.
- Enter the full package and class name of the plug-in class. Note that the method name is always “beforeRSSave”.
- The class must implements BeforeRSSave interface with the method beforeRSSave.
- Since this method is fired once for each rowset, looping is necessary to loop through the rowset.

Example: BeforeRSSave

```
/**************************************************************************/
package com.deltek.enterprise.extensions.xt_project_y;
import com.deltek.enterprise.system.applicationinterface.*;

public class MyPjmbasicBeforeRSSave implements BeforeRSSave{
    public String sProjId = "";
    /* Never commit in a beforeRSSave plug-in */
    public short beforeRSSave(ResultSetInterface rsI) throws DEException {
        SqlManager sqlMgr = rsI.getSqlManager(this);
        RowSetInterface roI = rsI.getRowSet();
       /* Looping is necessary */
        RSIterator iT = rsI.findInit(roI.ROW_New, roI.ROW_MarkDeleted,true);
        while (iT.next() != roI.UNDEFINED_CONTEXT) {
               sProjId = roI.getStringValue("PROJ___PROJ_ID");
        if (roI.isRowNew()) {
                    ..call some private method……
              }
        }
        return CP_OK;
    }
}
/**************************************************************************/
```

AfterRSSave

Timing

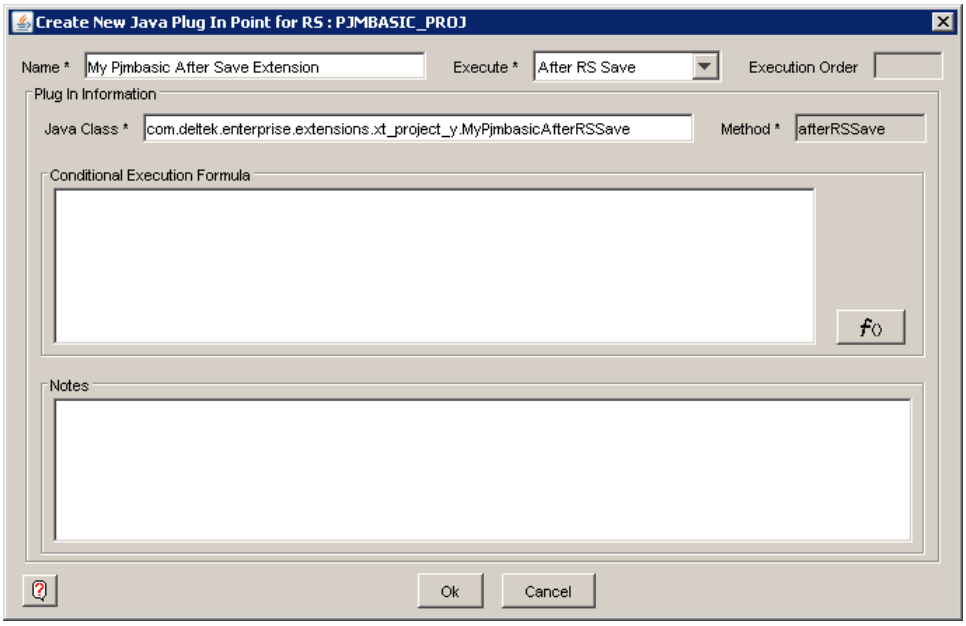
- AfterRSSave is invoked only during a save transaction. It is never invoked during data entry regardless of connection mode.
- After the BeforeRSSave and standard Save are executed, the system calls the AfterRSSave plug-in class.

Rules

- AfterRSSave is used to save data to tables other than the main table (if these tables are needed after the main table is saved). For example: Saving a new PROJ requires saving a new PROJ_EDIT.
- There should be no commit in this method since it is still within the transaction.
- Although not recommended, you can perform validation in this plug-in and if error severity message is added to the message container (via method addObjectMessage or addRowMessage, and so on) the framework will rollback all updates done so far in the save event.

- Tip: AfterRSSave can be used to select data updated in BeforeRSSave and standard Save if the it is more efficient with SQL than with java code. This is possible since the connection is within the same transaction. For example, select amount from header table and compare with total from all its children line.

Extensibility



- Enter a descriptive name for the plug-in. Select After RS Save.
- Enter the full package and class name of the plug-in class. Note that the method name is always “afterRSSave”.
- The class must implements AfterRSSave interface with the method afterRSSave.
- Since this method is fired once for each rowset, looping is necessary to loop through the rowset.

Example: AfterRSSave

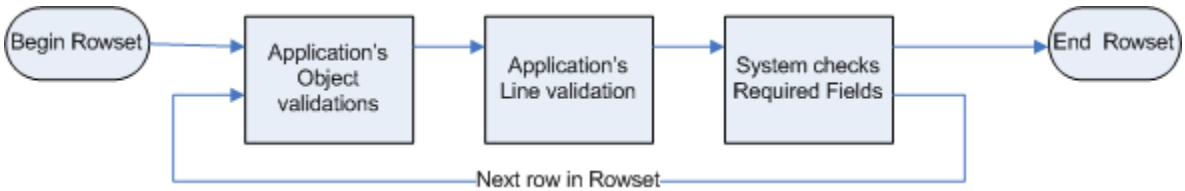
```
/*****************************************************************************/
package com.deltek.enterprise.extensions.xt_project_y;
import com.deltek.enterprise.system.applicationinterface.*;

public class MyPjmbasicAfterSave implements AfterRSSave {
    public String sProjId = "";
    public String sActiveFl = "";
    private String sOrigActiveFl = "";

    public short afterRSSave(ResultSetInterface rsI) throws Exception {
        /* Never commit in a afterRSSave plug-in */
        SqlManager sqlMgr = rsI.getSqlManager(this);
        RowSetInterface roI = rsI.getRowSet();
```

```
        sProjId = roI.getStringValue("PROJ___PROJ_ID");
        sActiveFl = roI.getStringValue("PROJ___ACTIVE_FL");
       /* Looping is necessary */
        RSIterator iT = rsI.findInit(roI.ROW_New, roI.ROW_MarkDeleted,true);
        while (iT.next() != roI.UNDEFINED_CONTEXT) {
            if (roI.isRowModified()) {
                sOrigActiveFl = roI.getStringValue("ORIG_ACTIVE_FL");
                if (!(sActiveFl.equals(sOrigActiveFl))) {
                  String sSql = "UPDATE PROJ_ORG_ACCT SET ACTIVE_FL = :sActiveFl " +
                      "WHERE PROJ_ID = :sProjId";
                  sqlMgr.SqlPrepareAndExecute(sSql);
                }
            }
        }
    sqlMgr.close();
    return CP_OK;
    }
}
/*************************************************************************/
```

Order on Save

Since there are many validation and save plug-in events, it is worthwhile to review the sequence.

Validation sequence (intra result set)

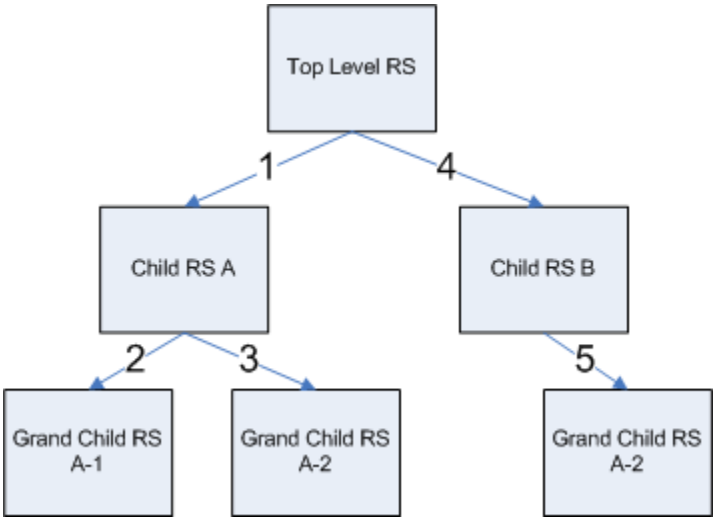


- For each line in a rowset, object validation is done first (in the order the objects are displayed in the DESC tab, unless overridden by the validation sequence).
- Row validation is done next.
- System checks for all required fields on that line
- Go to next row in rowset.
- When a rowset is done, system performs RS validation
- Then move to the next rowset until all the rowset for the RS is finished
- Example: Top level Result set has two rows: Row 1 and Row 2. Object Validation will be done for row 1, then Line Validation for row 1, and then Required fields for row 1. Moving to Row 2: Object

will be done for row 2, then Line Validation for row 2, then Required fields for row 2. When both rows are validated, then RS validation will be called.

Validation sequence (inter result sets)

- If there is more than one result set in an application, the system traverses down the tree and completes the intra RS validation (described above) one result set at a time.



- The order of the children selected (when they are at the same level) is not determinate. That is, there is no significance to choose Child RS A or Child RS B first.

Save sequence

- Once validation for all result sets is completed, the save transaction starts.
- Extensibility Before RS Save plug in is executed. If there are more than one result set, the inter result set set sequence is the same as described above.
- Then standard Deltek Before RS Save is executed (for all result sets).
- Then standard RS Save is executed. Delete is done first, then Update, then Insert.
- Then After RS Save is executed
- Then Extensibility After RS Save is executed.
- Save transaction completes.

Actions

An Action is a set of logic that is executed on the request of the user. It is in addition to the stadard event such as RS Populate, Validation or Save.

Usually action is triggered via application specific buttons on the application screen or via the drop down from the Gear icon on the main menu.

ActionInterface

Unlike other events where the plug-in class receives the handle to the ResultSetInterface, all action plug-in class receives the handle to the ActionInterface. From ActionInterface, you can obtain

ResultSetInterface and from there do your normal data retrieval logic. ActionInterface contains additional methods to handle long running process such as displaying progress meter dialog, controlling database transaction, and so on.

Here are some commonly used methods provided by the interface (see javadoc for more explanations)

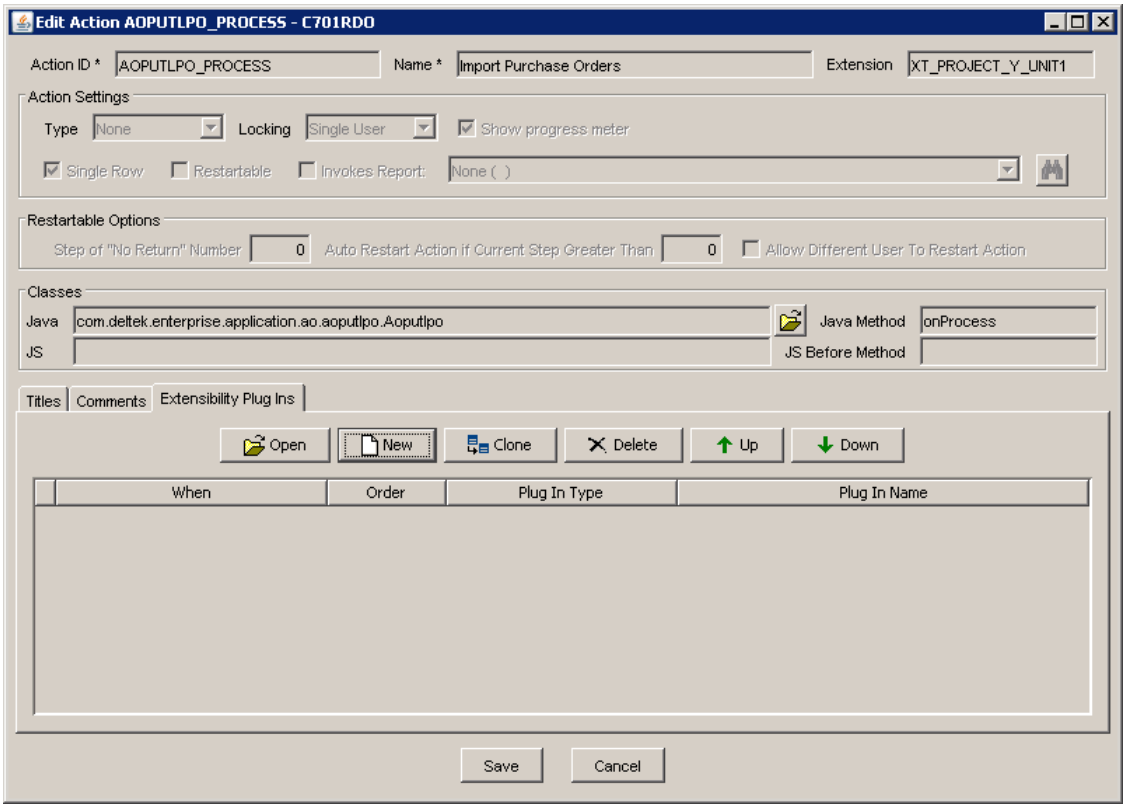
- **getResultSet()**: Return the ResultSetInterface of the screen the action is assigned to in the Designer. For a process, this is usually the parameter screen.
- **getApplication()**: Return the AppInterface for the application this action was invoked from.
- **getSqlManager()**: Return an instance of SqlManager
- **getFileManager()**: Return an instance of FileManager to access disk file

With Extensibility, you can extend an existing Deltek action or add a brand new action.

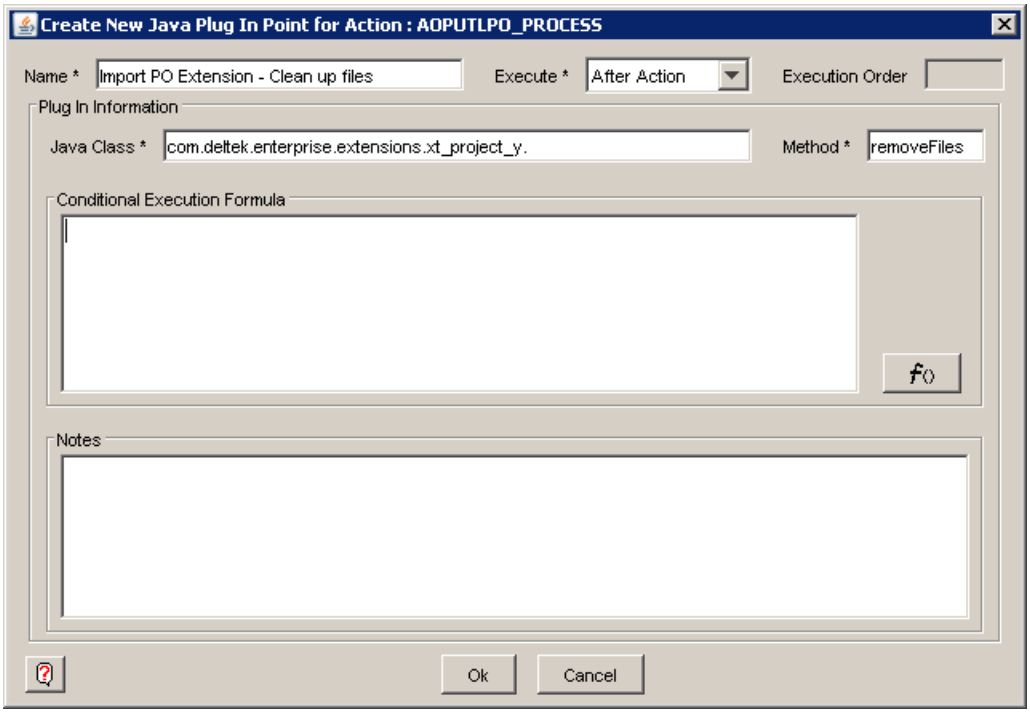
Web Services

If you are interested in invoking Costpoint Web Services from Extensibility plug-ins, refer to “Appendix C: Example of Invoking Generated Web Service from inside Costpoint Extensibility Code” in the *Deltek Costpoint Integration Overview* guide.

Extending a Standard Deltek Action



- In the Designer, select **Project » Unit » Extend Action**, and select action to extend.
- Select Extensibility Plug Ins tab and create the Java extension for the action.



- Enter a descriptive name for the plug-in. Select Before Action or After Action to be executed before or after the standard Deltek action logic.

> **Note:** If you add a Message with a Severity ERROR or FATAL inside your Before Action plug-in, all the subsequent Before Action plug-ins and main Action will be canceled. After Action plug-ins will be executed, regardless of the severity of the messages in the container. You can use the method `isProcessCanceled` in `ActionInterface` to check if the main process was canceled or not.

- Enter the full package and class name of the plug-in class. Note that the method name can be any name since we are not implementing any required interface like with RowValidation , RSValidation or Before or AfterRSSave. The method is public and return a boolean.

Example: Using FileHandlerInterface

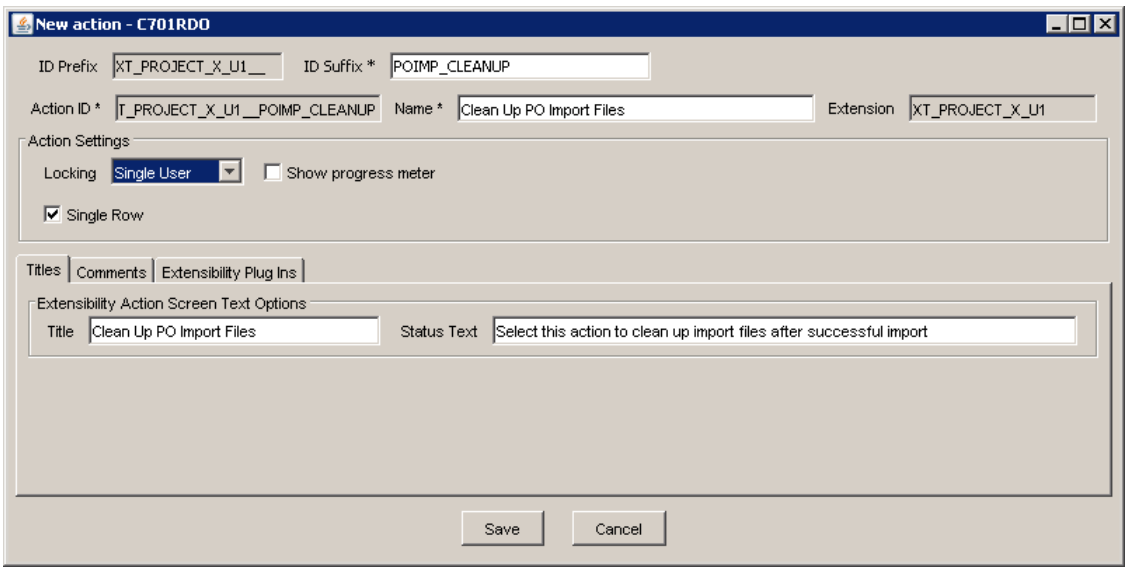
```
/*********************************************************************/
package com.deltek.enterprise.extensions.xt_project_y;
import com.deltek.enterprise.system.applicationinterface.*;

public class ImportPO {
    public boolean removeFiles (ActionInterface actI) throws DEException {
        ResultSetInterface rsI = actI.getResultSet();
        RowSetInterface roI = rsI.getRowSet();
        String fileName = roI.getStringValue("INPUT_FILE_NAME");
        FileHandlerInterface fileI = actI.getFileManager();
        if (!fileI.deleteFile(fileName)) {
```

```
            rsI.addObjectMessage("INPUT_FILE_NAME", "XT_PROJECT_Y_UNIT1__DEL_FILE_ERR",
rsI.ERROR);
            return false;
        }
        return true;
    }
}
/*************************************************************************/
```

Adding a new extensibility action

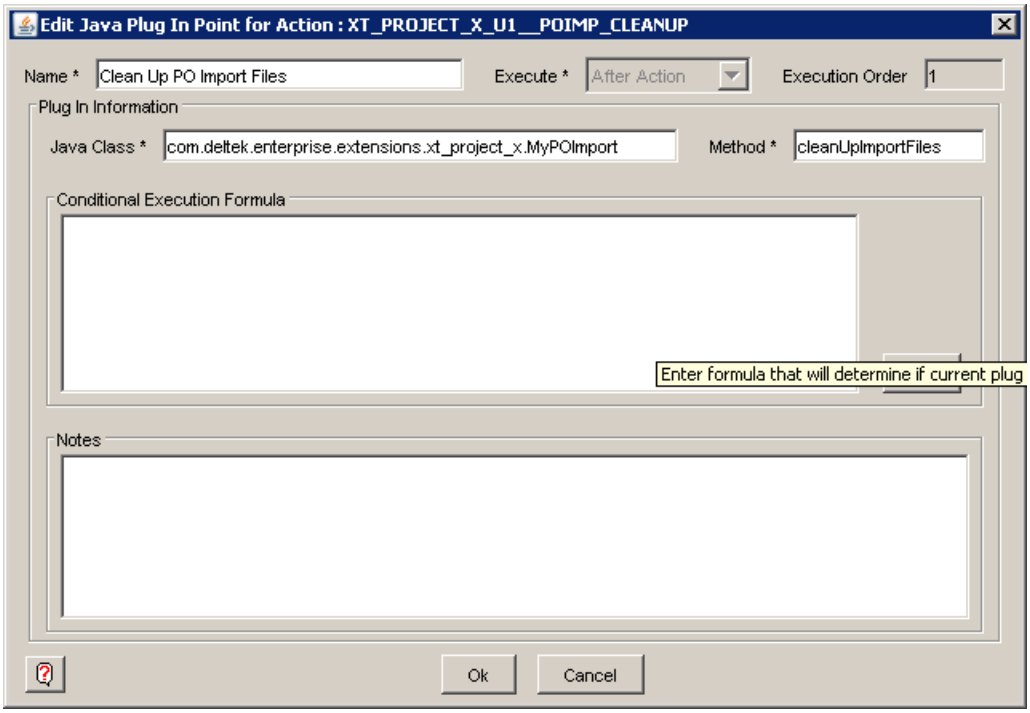
The steps to add new action to an existing application are very much the same as extending an existing Deltek action.

- In the Designer, select **Project » Unit » New » Action**.
- Define the action.



Attention: See Extensibility Designer User Guide for explanation of fields on this screen.

- Add Java plug-in for the action and specify the class name and method.



- Notice that is no selection for Execute mode. There is no before or after selection. Your java class plug-in executes by itself without any standard Deltek action class since the action is created by you.
- Once you have the new action defined, the java class is implemented normally like you have for extended action.

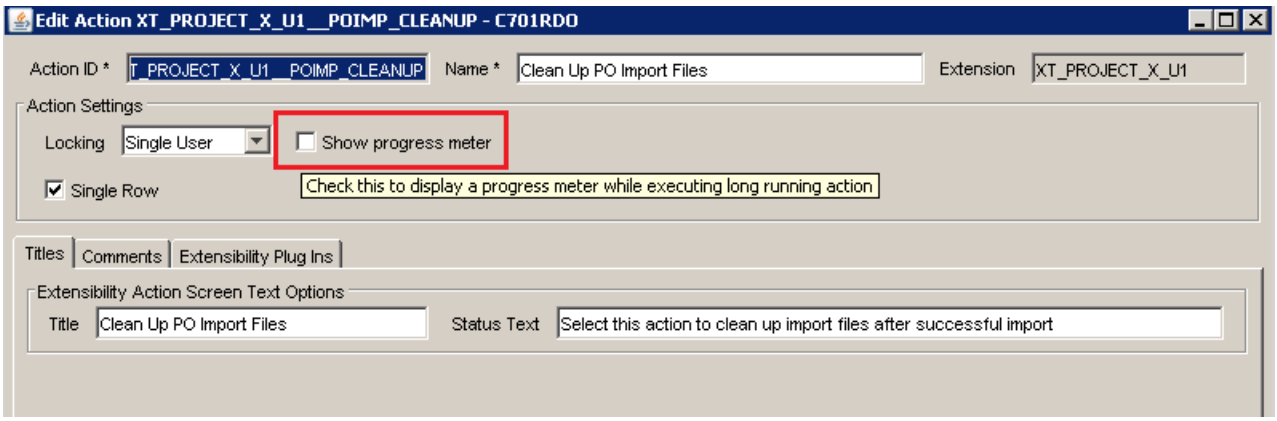
Example: Using FileHandlerInterface

```
/*****************************************************************************/
package com.deltek.enterprise.extensions.xt_project_y;
import com.deltek.enterprise.system.applicationinterface.*;

public class MyPOImport {
    public boolean cleanUpImportFiles (ActionInterface actI) throws DEException {
        ResultSetInterface rsI = actI.getResultSet();
        RowSetInterface roI = rsI.getRowSet();
        String fileName = roI.getStringValue("INPUT_FILE_NAME");
        FileHandlerInterface fileI = actI.getFileManager();
        if (!fileI.deleteFile(fileName)) {
            rsI.addObjectMessage("INPUT_FILE_NAME", "XT_PROJECT_Y_UNIT1__DEL_FILE_ERR",
rsI.ERROR);
            return false;
        }
        return true;
    }
```

```
}
/*************************************************************************/
```

Action Progress Meter

To show the action progress meter when an action is executed, the “Show Progress Meter” must be checked.



Text for Progress Meter

To enable internationalization of text shown on the progress meter, text must be entered in the Resource Message in the Designer. Then in application java code, use the appropriate method from ActionInterface to set the text by passing the Message ID (see below).

Methods for Progress Meter

Common methods for manipulating progress dialog are available from ActionInterface:

- void **setDlgMeterLimit**(int meterLimit): Sets the status meter limit
- void **SetDlgMeterValue** (int meterValue): Sets the status meter value
- void **setDlgCountText**(java.lang.String sText): Sets the action status count text property
- void **setDlgCount**(int nCount): Sets the action status count
- void **setDlgMeterTextLine**(int lineNo, java.lang.String meterTextID): Sets the status text line to the text ID in Resource Messages.

  LineNo = ActionInterface. DLG_TOP_LINE or ActionInterface.DLG_ BOTTOM_LINE

  Use DLG_TOP_LINE for major steps and DLG_ BOTTOM_LINE for minor steps.

  ```
  actionI.setDlgMeterTextLine (ActionInterface.DLG_TOP_LINE,
  "XT_PROJECT_X_UNIT1_TOP_TEXT");

  actionI.setDlgMeterTextLine (ActionInterface.DLG_BOTTOM_LINE,
  "XT_PROJECT_X_UNIT1_BOT_TEXT");
  ```

- boolean **checkUserCancel**(): Checks if the action is being cancelled by user. Returns TRUE if the action was cancelled, FALSE otherwise. Call this function as often as necessary to check if user has tried to cancel.
- public void **setDlgCancelText**(int optionNo): Set the confirm message to display when user cancels the action. Set optionNo to either CANCEL_TXT_HARD to display a stronger warning that some table have been updated or to CANCEL_TXT_SOFT for a normal confirmation text.

- void **addMessage**(String msgId,short msgType): Add messages to be displayed at the end of the action (or when action fails). These messages will be displayed at the bottom similar to validation messages. If any of the messages has severity of ERROR or higher, the system treats the process as un-successful.

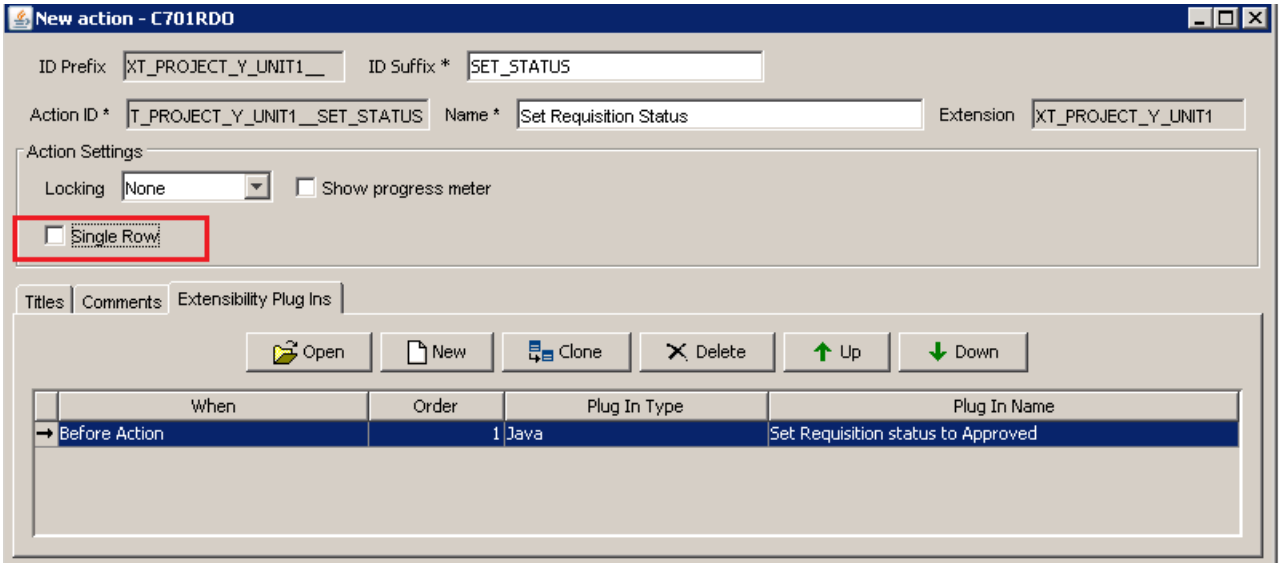
Example: Using progress meter

```
/*************************************************************************/
public class MyAction {
    private SqlManager sqm;
    private ActionInterface actI;
    public short processWithDialog (ActionInterface actionI) throws DEException,SQLException {
       actI = actionI;
       actI.setDlgMeterLimit(100);
       actI.setDlgMeterTextLine (actI.DLG_TOP_LINE, “CP_YDLG_PREPROC_DATA”);
       actI.setDlgMeterTextLine (actI.DLG_BOTTOM_LINE, CP_YDLG_CREATE_WRK_TBL);
       sqm = actI.getSqlManager(this);
       if (!doStep1())
           return false;
       if (!doStep2())
           return false;
       … more steps …
       actI.setDlgMeterValue(100);
       return true;
    }

    private boolean doStep1() throws DEException, SQLException {
       actI.setDlgMeterValue(10);
       if (funcCheckUserCancel())
           return false;
       … do Step 1 …
       sqm.SqlCommit();
       return true;
    }

    /* Check for user cancelling */
    private boolean  funcCheckUserCancel()  throws DEException{
       if (actI.checkUserCancel()) {
           actI.setDlgMeterTextLine (actI.DLG_TOP_LINE, “CP_YDLG_PROC_CANCEL”);
              actI.setDlgMeterTextLine(actI.DLG_BOTTOM_LINE,"");
              return true;
```

```
        }
        return false;
    }
}
/*************************************************************************/
```

Maintenance Action

Action can be used on a maintenance / transaction application to aid in faster data entry. These generally do not process or update data in the database. They are mainly used for defaulting or setting values to the data user has on the screen. These are considered maintenance action. They are not long running action.



Single Row is usually checked for long running action and generally is not checked for a maintenance action. When it is not checked, the code needs to loop through the rows in the result set.

Example: Loop through rows selected by user and change Requisition status to Approved

```
/*************************************************************************/
package com.deltek.enterprise.xt_project_y.req;
import com.deltek.enterprise.system.applicationinterface.*;

public class RqLineAction {
    public boolean setStatus (ActionInterface acI) throws DEException {

        /* get ResultsetInterface from ActionInterface */
        ResultSetInterface rsI = acI.getResultSet();
        RowSetInterface roI = null;
       /* Create iterator that looks for row selected and exclude row mark deleted */
        RSIterator rst = rsI.findInit(roI.ROW_Selected,roI. ROW_MarkDeleted, true);
```

```
        roI = rsI.getRowSet();

        /* Loop through rows and set values */
        while (rst.next()!= roI.UNDEFINED_CONTEXT) {
            roI.setStringValue("A","S_RQ_STATUS_CD");
        }
        return true;
    }
}
/*************************************************************************/
```

About Deltek

Better software means better projects. Deltek delivers software and information solutions that enable superior levels of project intelligence, management and collaboration. Our industry-focused expertise makes your projects successful and helps you achieve performance that maximizes productivity and revenue. www.deltek.com