Developer's Guide

Version 8.04

September 2006 — Second edition

Revision date	Edition	Comments
January 2006	First edition	First edition for version 8.0.
September 2006	Second edition	Updated as of patch 804.

The developer's guide is intended for developers who need to use the Calypso API to build custom components, or to customize and extend existing functionality. Extension capabilities are described through a comprehensive collection of examples.

Refer to the class library documentation on the support web site (also referred to as Javadoc) for a comprehensive description of all the classes and methods mentioned in the examples.

Contents

Section 1. Getting Started

1.1 How to Compile and Execute Samples

1.1.1 Compiling

To compile the samples located in the `samples` package, go to $CALYPSO_HOME or to the root directory where the Calypso system resides, and type the following at the command line (this is a one-line command):

```
javac -d ./build -classpath "./jars/calypso.jar" -sourcepath ./src src/samples/<class name>.java
```

For example:

```
javac -d ./build -classpath "./jars/calypso.jar" -sourcepath ./src src/samples/LoadTrade.java
```

To compile the samples located in the `calypsox` package, rename the classes from Sample<ClassName> to <ClassName>, go to $CALYPSO_HOME or to the root directory where the Calypso system resides, and type the following at the command line (this is a one-line command):

```
javac -d ./build -classpath "./jars/calypso.jar" -sourcepath ./src src/calypsox/<the fully qualified name of the class>.java
```

1.1.2 Executing

Prior to executing the samples, you need to populate the database with sample data using the database scripts located in `samples/cookbook/sql/cookbook_sybase.sql`.

To execute any of the samples, type the following at the command line from $CALYPSO_HOME/build (this is a one-line command):

```
java <the fully qualified name of the class> -env <envName> –user <username> -password <passwd> (+ any required arguments)
```

For example:

```
java samples/TestCon –env myEnv -user calypso_user –password myPassword
```

To find out what arguments a program needs, simply run the program with no argument and the program will list the required arguments.

In all the samples we will assume there is a Calypso environment name myEnv, a Calypso user name calypso_user, and password myPassword. You can specify your Calypso settings by clicking the ... button in AppStarter as shown below.

1.2 How to Create a Custom Package

All calypso files are placed under the `com.calypso` package. You can place any extension to the system under the `calypsox` package. The system by default will search for new files in that package.

Alternatively you can create a custom package name and place your extension in your own package. The following sample illustrates the steps required to create your custom package.

Do the following for creating a custom package:

1. Create a directory for the custom package.

Note - If the custom package directory is not placed under $CALYPSO_HOME then you will need to modify your class path to include the location of the directory.

2. Create a class named `calypsox.tk.util.CustomGetPackages` that implements the interface `com.calypso.tk.util.GetPackages`.

Implement the *getPackages()* method . The method should return a vector containing the names of all the packages you want to add.

This class will be invoked from `com.calypso.tk.util.InstantiateUtil`.

3. Restart all Calypso engines and applications.

Tip

- You can choose the logging category "Instantiate" to debug class instantiation.

Sample Code

In this sample, we have created a package called `samples.cookbook`.

» Sample in `calypsox/tk/util/CustomGetPackages.java`

```
package calypsox.tk.util;

import com.calypso.tk.util.GetPackages;
import java.util.*;
import java.lang.String;

/**
 * A utility interface to search for the particular package.
 * That class will be used to get the Packages in Calypso.
 */
public class CustomGetPackages implements GetPackages  {
    public Vector getPackages() {
        Vector v = new Vector();
        v.addElement("samples.cookbook");

        return v;
    }
}
```

1.3 How to Create Custom Code

Custom code should be located under a `$CALYPSO_HOME/custom` directory with the proper package name, either `calypsox` for example `custom/calypsox/tk/service/CustomDSStartUp`, or your own custom package name.

The `$CALYPSO_HOME/src/calypsox` directory is reserved for sample custom code provided by Calypso.

Details are described in `$CALYPSO_HOME/custom/ReadMe.txt`.

1.4 How to Create a Client Application

1.4.1 Connecting to Data Server and Event Server

Anytime you write a client application that requires access to Calypso data such as domain data (currency code list, reference index list, etc.) and persistent data (trades, curves, positions, etc.), the client application needs to

establish a connection to the Data Server. Furthermore if a client application needs to publish or subscribe to Calypso events, it needs to establish a connection to the Event Server.

Once the connection is established, you can access Calypso data using the data services.

The example below illustrates how to create a connection to the Data Server and Event Server and then disconnect before the program exits.

Do the following for connecting to Data Server and Event Server:

1. Use *com.calypso.tk.util.ConnectionUtil::connect* to create a Data Server connection. This method returns a `com.calypso.tk.service.DSConnection` object.

2. Use *com.calypso.tk.event.ESStarter::startConnection* to create an Event Server connection. This method returns a `com.calypso.tk.event.PSConnection` object.

Note - The PSConnection object returned from *ESStarter.startConnection()* does not establish the physical connection between the client application and the Event Server. The *start()* method in PSConnection must be called to establish the actual connection.

3. Use *PSConnection::stop* to disconnect from the Event Server.

4. Use *DSConnection::disconnect* to disconnect from the Data Server.

Tips

- The *connect* method in ConnectionUtil is overloaded to allow using different arguments and options to create a connection to the Data Server. Refer to `com.calypso.util.ConnectionUtil` for details. If you do not want to use ConnectionUtil, you may directly use the *connect* method in DSConnection.
- The *startConnection* method in ESStarter is overloaded to allow using different arguments and options to create a connection to the Event Server. Refer to `com.calypso.event.ESStarter` for details.
- Once connected, the DSConnection object will contain all the settings in your Calypso environment. You can access those settings using the different *get* methods in DSConnection. The static method *DSConnection.getDefault* returns the connection created, thus preventing you to have to pass or store the DSConnection object in your code.
- The static methods *PSConnection.setCurrent* and *getCurrent* allow you to store the PSConnection created in your client application, thus allowing your client application to access it anywhere.

Sample Code

» Sample in `samples/TestCon.java`

```
package samples;

import com.calypso.tk.util.*;
import com.calypso.tk.service.*;
import com.calypso.tk.event.*;
import com.calypso.tk.core.*;
import com.calypso.tk.bo.*;
import java.util.*;

/**
 * Connects to the data server and event server and then disconnects.
*/
public class TestCon {
    static public void main(String args[]) {

       DSConnection ds = null;
       PSConnection      ps=null;

       try { // Starts connection to DataServer
        ds = ConnectionUtil.connect(args, "TestCon");
       }
       catch (Exception e) {
           Log.error(Log.CALYPSOX, e);
           return ;
       }
       try { //Starts connection to EventServer
```

```
            // The second argument is only necessary for subscribing to events.
            ps = ESStarter.startConnection(ds, null);
            ps.start();
            PSConnection.setCurrent(ps);
        }
       catch (Exception e) {}

       System.out.println("Calling PS stop "); // Close connection to Event
Server
       try {PSConnection.getCurrent().stop();}
       catch (Exception e) {Log.error(Log.CALYPSOX, e);}
       System.out.println("Calling PS stop done ... ");

       System.out.println("Calling DS Disconnect "); // Close connection to Data
Server
       DSConnection.getDefault().disconnect();
       System.out.println("Calling DS Disconnect done ... ");
    }
}
```

1.4.2 Handling a Lost Connection to the Data Server

When writing a client application you may want to be notified and subsequently attempt to reconnect if the connection to the Data Server is lost. The Data Server provides the following mechanisms to auto-reconnect when the connection is lost.

- Using *DSConnection.setAutoReconnect()*
- Implementing ConnectionListener

Using DSConnection.setAutoReconnect()

Set *setAutoReconnect()* to true and set the following parameters as applicable:

- RETRY = Number of times you want to try to auto-reconnect.
- TIMEOUT_RECONNECT = Time after which you try to reconnect.

Implementing ConnectionListener

Do the following for implementing ConnectionListener:

1. The client application needs to implement `com.calypso.tk.service.ConnectionListener`
2. The client application has to be registered with the Data Server by calling *DSConnection::addListener(…)*

Sample Code

The following example illustrates how to extend the client application in the previous section to auto-reconnect to the Data Server.

» Sample in `samples/UseDSListener.java`

1.4.3 Creating Custom Initialization Code

The Data Server provides the following mechanisms for invoking custom code:

- Using DSStartUp for invoking initialization code when the Data Server is started
- Using DSConnectionInit for invoking initialization code on the client, each time a DSConnection is created

Using DSStartUp

Your code will be invoked when the Data Server is started after the SQL init but before the start of the RMI Services (i.e. before any client can connect). One usage of this mechanism is to allow the Data Server to preload data that will be frequently accessed by client applications. For example, you can preload certain books in the system. This will prevent the Data Server from having to retrieve the trades for those books individually.

Create a class named `tk.service.CustomDSStartUp` that implements the interface `com.calypso.tk.service.DSStartUp`, and implement your code in the *onStartUp()* method.

This class will be invoked from `com.calypso.tk.service.DataServer`.

Sample Code

- Sample in `calypsox/tk/service/CustomDSStartup.java`. This sample loads PricingEnv and TradeFilters set by properties PRICING_ENV_STARTUP and TRADE_FILTER_STARTUP.

Using DSConnectionInit

To specify a custom initialization class for client applications which will be called after a DSConnection is created and started.

Create a class named `tk.service.CustomDSConnectionInit` which implements the interface `com.calypso.tk.service.DSConnectionInit`.

This class will be invoked from `com.calypso.tk.service.DSConnection`.

Sample Code

- Sample in `calypsox/tk/service/CustomDSConnectionInit.java`

1.4.4 Customizing the RMI Socket Factory

You can create your own RMI Socket Factory, to specify for example, a unique port for RMI communication in case of a firewall.

Create a class named `tk.service.CustomDSSocketFactory` which implements the interface `com.calypso.tk.service.DSSocketFactory`.

This class will be invoked from `com.calypso.tk.service.DSConnection`.

Sample Code

- Sample in `calypsox/tk/service/CustomDSSocketFactory.java`

1.4.5 Creating a User Startup Routine

This allows you to implement your own class to do things like pre-load C++ libraries, which would be useful, for example, for Scheduled Task EOD_VALUATION.

Create a class named `apps.main.UserStartup`.

This class will be invoked from `com.calypso.apps.startup.AppStarter`.

Section 2. Data Services

2.1 How to use the Data Server

The Data Server is a single point of access for all Calypso data. No client application should ever access the database directly. Once you have a connection to the Data Server as described above, the Data Server is accessed through a set of remote services located under the `com.calypso.tk.service` package, each responsible for a different group of data:

- RemoteMarketData handles pricing information such as interest rate curves
- RemoteReferenceData handles static data such as counterparty definitions
- RemoteProduct handles financial instrument definitions such as futures contracts
- RemoteTrade handles trade information
- RemoteAccess handles the access permission and system security data
- RemoteAccounting handles the accounting rules data
- RemoteBackOffice handles back office specific data

The Calypso online documentation provides detailed descriptions of the objects handled by each service under `com.calypso.tk.service`.

The Data Server in turn will access the database or the data stored in cache. The caches maintained by the Data Server are configured and administrated from the Calypso Administrator window. However, for data that is frequently accessed such as reference data, it is more efficient to cache those data locally in the client. The BOCache and LocalCache classes are provided for that purpose. It is their responsibility to access the Data Server.

Hence, when you want to access Calypso data:

1. Check whether BOCache or LocalCache handle those data and use BOCache and LocalCache.

2. If BOCache or LocalCache do not handle those data, then use the appropriate remote service.

2.2 How to use Local Cache

LocalCache allows client applications to maintain client caches for a number of static data including Currency Indexes, Rate Indexes, Rate Index Defaults, Currencies, Domains, and FX resets, thereby enhancing the performance.

When data is retrieved using LocalCache, if the data is not available in the cache, LocalCache will retrieve the data from the data server and update the cache, making the data available for subsequent requests.

For caching other types of static data, use BOCache.

Tips

- You will be able to maintain the cache consistency by having an instance of `tk.util.CacheConnection`. This class keeps the cache consistent by subscribing to PSEventDomainChange, PSEventQuote, PSEventQuoteRemoved, and PSEventCreditRating events. It also updates the cache according to the timer specified in the Admin window.

```
CacheConnection cacheConnection = new CacheConnection(DSConnection.getDefault());
```

- When the client application needs to modify domain values for display purposes for example, you need to clone the data using *cloneDomainValues()*, and modify the cloned data. Do not modify the returned list directly otherwise the master list in LocalCache will be changed.

- For retrieving static data not handled by LocalCache or BOCache use `tk.service.RemoteReferenceData`.

- Do not use LocalCache for code that will be executed within the Data Server.

See also

- `com.calypso.apps.util.AppUtil` class for helpful object loading methods.
- Refer to `com.calypso.tk.service.LocalCache` for available methods and details.

Sample Code

The following sample illustrates how to use LocalCache in a client application.

It creates a connection to the Data Server, and alls the appropriate static method on LocalCache to retrieve the values of the "Principal Structure" domain without the Mortgage value.

» Sample in `samples/UseLocalCache.java`

2.3 How to use BOCache

BOCache allows client applications to maintain client caches for a number of static data including Accounts, Books, Contacts, Legal Entities, Settlement and Delivery Instructions, Exchange Traded Products, and Netting Configurations. BOCache also allows maintaining client caches for quotes.

When data is retrieved using BOCache, if the data is not available in the client cache, BOCache will retrieve the data from the data server and update the client cache, making the data available for subsequent requests.

For example, for a trade load, you would use BOCache to load the relevant TradeFilter to be passed to the *getRemoteTrade().getTrades(TradeFilter, Jdatetime)* method.

```
TradeFilter tf = BOCache.getTradeFilter(ds, "MyTradeFilter");
```

For caching other types of static data, use LocalCache.

Tips

- You will be able to maintain the cache consistency by having an instance of `tk.util.CacheConnection`. This class keeps the cache consistent by subscribing to PSEventDomainChange, PSEventQuote, PSEventQuoteRemoved, and PSEventCreditRating events. It also updates the cache according to the timer specified in the Admin window.

  ```
  CacheConnection cacheConnection = new CacheConnection(DSConnection.getDefault());
  ```

  A sample program `samples/TestMultiThreadQuotes.java` illustrates how to get refreshed quotes from BOCache.

- For retrieving static data not handled by BOCache or LocalCache refer to `tk.service.RemoteReferenceData`. See also Extending BOCache below.
- Do not use BOCache for code that will be executed within the Data Server.

Extending BOCache

If the data you want to access is not handled by BOCache or LocalCache, and you want to cache that locally, you can extend BOCache by providing an implementation of CustomClientCache in `tk.bo.CustomClientCacheImpl`.

This class will be invoked from BOCache.

You also have to publish a PSEventDomainChange for modified data so that BOCache will publish updates to CustomClientCacheImpl. PSEventDomainChange has a set of static integers to identify what data has changed (legal entity, book etc.). So you have to add your own set of static integers to identify changes specific to CustomClientCacheImpl.

Create a class named `tk.event.PSEventDomainChangeCustom` that extends `com.calypso.tk.event.PSEventDomainChange` and define integers for each custom data as in the example shown below.

```
final static public int MY_DATA1 = ID_MAX+1;
final static public int MY_DATA2 = ID_MAX+2;
```

Then publish a PSEventDomainChangeCustom for each custom data.

- Samples in `calypsox/tk/bo/CustomClientCacheImpl.java` and `calypsox/tk/event/PSEventDomainChangeCustom.java`

See also

- `com.calypso.apps.util.AppUtil` class for helpful object loading methods.
- Refer to `com.calypso.tk.bo.BOCache` for all available methods and details.

2.4 How to use a Remote Service

The following example demonstrates how to get a trade and product from the Data Server and subsequently save them as a new trade and product.

Do the following for using a remote service:

1. Create a connection to the Data Server.

2. Obtain the appropriate remote object from the Data Server connection.

3. Use the appropriate method in the remote object to retrieve or save the desired data.

Tip

- For objects which contain a unique id, this id is assigned by the Data Server when the object is first saved. The save methods in the remote services for those objects will return an assigned id if the save is successful. It is important to set the id returned to the object after the save is performed otherwise the object will continue to have a null id and saving the object again subsequently will result in a new copy of the object.

Sample Code

- Sample in `samples/cookbook/UseDataServer.java`
- For example, if you wish to output all of the ids of the trades in a TradeFilter, you would call *getTrades(TradeFilter, JDatetime)* on RemoteTrade to return all the trades associated with the TradeFilter and whose trade date is before the JDatetime.

```
DSConnection ds = null;
try {
      ds = ConnectionUtil.connect(args, "UseRemoteBO");
}
   catch (ConnectException e) {
      Log.error(Log.CALYPSOX, e);
      System.out.println("ERROR: Connection to data server failed.");
      System.exit(-1);
       }

JDatetime now = new JDatetime();
try {
      TradeArray v = ds.getRemoteTrade().getTrades(tradeFilter, now);

      for(int i=0;i<v.size();i++) {
      Trade trade = (Trade) v.elementAt(i);
      System.out.println("Trade : " + trade.getId());
      }
}
   catch (Exception exc) {}
```

- If you wish to retrieve a price quote for a product, you would use the RemoteProduct interface to load the product, and the RemoteMarketData interface to load the quote. The *getProduct(int)* method of RemoteProduct returns the product for a given Product id. The *getQuoteValue(QuoteValue)* method of RemoteMarketData returns a QuoteValue object which contains the quote value and type.

```
DSConnection ds = null;
try {
      ds = ConnectionUtil.connect(args, "UseRemoteBO");
}
```

```
   catch (ConnectException e) {
      Log.error(Log.CALYPSOX, e);
      System.out.println("ERROR: Connection to data server failed.");
      System.exit(-1);
       }

Product product = null;

try {
      product = ds.getRemoteProduct().getProduct(inputProductId);
}
      catch (Exception exc) {}

JDatetime now = new JDatetime();
JDate quoteDate = now.getJDate(TimeZone.getDefault());

// Initialize a QuoteValue object
// use quote type NONE - will be set by getQuoteValue method
QuoteValue q = new QuoteValue(quoteSetName, product.getQuoteName(),
      quoteDate, QuoteValue.NONE, 0, 0);
try {
      q = ds.getRemoteMarketData().getQuoteValue(q);
}
   catch (Exception exc) {}
```

Complete sample in samples/QuoteLoaderSample.java.

2.5 How to Extend the Data Server

When a custom object that requires persistence is added to the system, the Data Server has to be extended to support the saving/loading and caching of the object. If necessary, the extension should also support event publishing when the object is first saved or modified.

The Calypso API provides two ways to extend the Data Server:

- Using the DSTransactionHandler, or
- Writing a custom remote service.

The DSTransactionHandler is intended to accommodate small number of custom objects. As illustrated in the next section, each custom object requires creating its own set of DSTransactionHandler and other related classes. Doing so for a large number of custom classes it might prove cumbersome but for a small number of classes this it is an efficient mechanism to extend the Data Server.

Note - The extension of the Data Server is only required for custom objects that do not extend from an existing persistent Calypso object. The system has other mechanisms to support children of persistent Calypso objects and the dedicated mechanisms should be used instead. For example adding a new product or market data does NOT require extending the Data Server.

The following sections will demonstrate how to extend the Data Server using both methods. In the examples we will extend the Data Server to handle the persistency, caching and event publishing of a custom "equity basket".

2.5.1 Persistence and Caching

Persistence

To handle the persistence of the equity basket class it is necessary to write an SQL class and create a database table. These steps are required regardless of which extension method is used.

Sample Code

» Sample for EquityBasket in samples/cookbook/tk/product/EquityBasket.java

Note - The EquityBasket class is for illustration purposes only.

» Sample for EquityBasketSQL in samples/cookbook/tk/product/sql/EquityBasketSQL.java

» Sample for database scripts in `samples/cookbook/sql/cookbook_sybase.sql`

Tips

- **Error Handling** — The SQL methods should throw a PersistenceException so that the user of the client application will receive an error message if an SQL error occurs. The remote methods that call your SQL methods will catch any thrown PersistenceException and create and throw a RemoteException containing that PersistenceException.

  For example:

```
void sqlfoo() throws PersistenceException {
   try {
      ...
   }
   catch (SQLException e) {throw new PersistenceException(e.getMessage()):}
}
```

  The typical RMI method that calls an SQL method will handle the error as follows:

```
void rmiCall() throws RemoteException {
   try {
      sqlfoo();
   }
   catch(Exception e) {throw new RemoteException(e.getMessage());}
}
```

- **JResultSet** — Calypso recommends using `com.calypso.core.sql.JResultSet` to work with query results when you retrieve multiple records from the database. JResultSet is a wrapper class (around a ResultSet object) that adds the methods *getJDate()* and *getJDatetime()* to return a date or datetime from any cell in a table of query results.

  The example below shows how one might use a JResultSet to compile a Vector of trades with their Trade Id numbers and Trade Date/Settle Date stamps. In this example, all Trade Date and Settle Date stamps will be expressed in the system's local time zone:

```
static public Vector getAllTradeTimestamps() {
   Vector tradeVector = new Vector();
   Connection con = ioSQL.newConnection();
   Statement stmt = null;
   try {
      stmt = ioSQL.newStatement(con);
      JResultSet rs = new JResultSet(stmt.executeQuery("SELECT \
         trade id, trade date time, settlement date FROM trade,book \
         where book.book name='TRADINGC' and trade.book id=book.book id"));
      int j;
      while(rs.next()) {
         j=1;
         Trade trade = new Trade();
         tradeVector.addElement(trade);
         trade.setId(rs.getInt(j++));
         trade.setTradeDate(rs.getJDatetime(j++));
         trade.setSettleDate(rs.getJDate(j++));
      }
      rs.close();
   }
   catch( Exception e ) { display(e); }
   finally {
      try {ioSQL.close();}
      catch (Exception e) {}
      ioSQL.releaseConnection(con); }
   return tradeVector;
}
```

- **Dates** — Calypso distinguishes between timestamps stored in the database and those displayed to users. Saved timestamps are expressed in the designated reference time zone, while timestamps displayed to and set by users are expressed in the local time zone, which is the time zone of the user's workstation. By default, the reference time zone for stored timestamps is GMT.

When using SQL queries to retrieve information from the Calypso database, keep in mind that all dates and times are expressed in the system's designated reference time zone. Thus any dates and times in your WHERE clauses must be expressed in the reference time zone, and you must convert all dates/times in your query results to the desired local time zone.

Calypso provides a set of methods to do this. You will find these methods in the class `com.calypso.tk.core.Util`. To convert a local date to a String for use in a WHERE clause, use the method *date2SQLString()* or *datetime2SQLString()*.

Alternatively, you can use *Util.ReferenceTZ2Local(Date)* to convert a date from the reference time zone to the local time zone. For details, see the online class documentation for `com.calypso.tk.core.Util`.

- **Commit and Rollback** — Whenever you have a commit in an SQL file, make sure to also have a rollback as shown in the example below.

```
Void save(myObject){
try{
   Connection con=ioSQL.newConnection();
   save(myObject, con);
   commit(con);
   Update the Cache or any hash Only after commit
}
catch(PersistenceException e){
   rollback(Con);
   Log.error(e,e)
}
finally{ releaseConnection(con)}
}
```

Note - Do not put the commit inside *save(myObject, con)*.

Caching

For objects that are frequently accessed it would be logical to cache those objects so that the Data Server does not have to retrieve them from the database. Refer to Cache Framework for details.

2.5.2 Using DSTransactionHandler and DSTransactionInput

The mechanism in the Data Server that accommodates custom objects involves the DSTransactionHandler and DSTransactionInput classes in the `com.calypso.tk.service` package. Custom DSTransactionHandler and DSTransactionInput classes for the object need to be added to the system. An SQL class that contains the persistency code for the object is also required. A client application that needs to access the custom object will create an instance of the custom DSTransactionInput class and then pass it to the Data Server. The Data Server will internally invoke the custom DSTransactionHandler which in turn uses the SQL class of the object to perform persistence and caching.

Overview of Steps

- Step 1 — Create an SQL class for the object as described under Persistence and Caching
- Step 2 — Create a DSTransactionInput class
- Step 3 — Create a DSTransactionHandler class

Step 2 — Creating a Custom DSTransactionInput

Create a class named `tk.service.<object name>TransactionInput` that extends `com.calypso.tk.service.DSTransactionInput`.

The *getHandler()* method specifies the name of the associated handler class, and the member variable transactionType specifies the task that the client program wants the Data Server to perform.

Sample Code

» Sample in `samples/cookbook/tk/service/EquityBasketTransactionInput.java`

Step 3 — Creating a Custom DSTransactionHandler

Create a class named `tk.service.<object name>TransactionHandler` that extends `com.calypso.tk.service.DSTransactionHandler`.

The handler class invokes the object's SQL class to perform the necessary persistence tasks. It also publishes an event after the save is completed.

This class will be invoked from `com.calypso.tk.service.AccessServerImpl`.

In this sample, we are treating the event as a persistent event. As such, the handler class will save the event to the database.

Note - PSEventEquityBasket is a new event type. See How to Create a Custom Event for details on creating new event types.

Tips

- In order for the Data Server to properly handle a custom persistent object, the class must implement the Serializable interface.
- When publishing a persistent event inside the Transaction Handler, the event MUST be published and saved within the same database transaction that the persistent object is handled, in order to ensure transactional atomicity. This way the event will not be published if any error is encountered while saving the object and the event.

Sample Code

- Sample in `samples/cookbook/tk/service/EquityBasketTransactionHandler.java`
- A sample client application in `samples/cookbook/UseEquityBasket.java` illustrates how to use DSTransactionInput to access an EquityBasket object from the Data Server.

2.5.3 Creating a Custom Remote Service

To create a custom Remote Service, an interface that extends `java.rmi.Remote` and a service class that implements this interface need to be added. In the following example we will add a remote interface RemoteCustomData and an implementation of the interface CustomDataServer. The example will also illustrate how to register CustomDataServer with the Data Server.

Overview of Steps

- Step 1 — Create an SQL class for the object as described under Persistence and Caching
- Step 2 — Create a remote service interface that contains methods to save, load and remove custom objects
- Step 3 — Create a server class that implements the remote service interface
- Step 4 — Register the custom server with the Data Server

Step 2 — Creating a Custom Remote Service Interface

Create a class named `tk.service.Remote<name>` that extends `java.rmi.Remote`.

Sample Code

The sample contains methods to handle the EquityBasket class. A similar set of methods can be added to the interface for any new custom object.

- Sample in `samples/cookbook/tk/service/RemoteCustomData.java`

Step 3 — Creating a Custom Remote Server

Create a class named `tk.service.<name>Server` that implements the custom remote service interface created in Step 2 above.

The service class is responsible for the following:

- Perform the requested persistence task by calling the appropriate method in the object's SQL class
- Publish (and save if necessary) events triggered by the saving and removing of an object

Sample Code

» Sample in `samples/cookbook/tk/service/CustomDataServer.java`

Step 4 — Registering the Custom Remote Server

Do the following for registering the custom server with the data server:

1. Create a class named `tk.service.CustomDSInit` that implements `com.calypso.tk.service.DSInit`.

Implement the *getServers()* method on this class. This method should return a Hashtable containing pairs of "name of server" / "instance of server".

This class will be invoked from `com.calypso.tk.service.DataServer`.

2. Compile CustomDSInit and restart the Data Server.

Tip

When the server class (implementation of the remote service) is compiled for the first time or when any of its API is changed, you will need to run `rmic` on the server class. To run `rmic` on a server, first compile the server class and then run the following at the command line:

```
rmic <fully qualified class name of the server class>
```

Sample Code

» Sample in `samples/cookbook/tk/service/CustomDSInit.java`

» To use the custom remote service, simply call *DSConnection::getRMIService(<service name>)*. Sample in `samples/cookbook/UseEquityBasket2.java` to see how to access EquityBasket objects using RemoteCustomData.

2.6 How to use the Data Server in Read-Only Mode

The read-only Data Server provides a mechanism to off load work from the active Data Server for items that are read tasks, such as generating reports. The activity may be high and could interfere with the normal usage of the Data Server.

Note - In read-only mode, the Data Server does not update the cache. The way that it is kept up to date is to listen to events that are sent by the active Data Server and to reload the information from the database as needed. This explains the requirement for the database connectivity in the read-only Data Server.

The call to enable this form of update, using events and not through the RMI services, is controlled by *setUseCacheSubscriber()*. It will be set to true if the DS_READ_ONLY property has been set to true, indicating that the Data Server is running in read-only mode. Note that this must be set before the Data Server is started.

Refer to the *Calypso System Guide* for complete information on setting up the Data Server in read-only mode.

How to Customize the Testing of SQL Queries

You can customize the testing of SQL Queries to see if they are allowed on a read-only Data Server.

Create a class named `tk.core.sql.CustomReadOnlySQLTest` that implements the interface `com.calypso.tk.core.sql.ReadOnlySQLTest`.

This class will be invoked from `com.calypso.tk.core.sql.CalypsoStatement`.

Section 3. Event Services

Calypso events are handled by the Event Server using a publish and subscribe mechanism.

3.1 How to Subscribe to and Publish Events

Once you have a PSConnection to the Event Server as describe above, you can subscribe to and publish events through event classes. PSEvent (`com.calypso.tk.event.PSEvent`) is the base class for all event types. Each derived class represents a specific type of event. For example, PSEventTrade models the events published when a user performs an operation on a trade.

Note - All events are named PSEvent<event type> and located under `com.calypso.tk.event`.

Events that require guarantee delivery are persistent. Hence events are classified into non-persistent and persistent events for processing. Persistent events have an SQL class `com.calypso.tk.event.sql.PSEventSQL`.

3.1.1 Publishing Events

Once you have a PSConnection, you can do the following:

- Publish non-persistent events using the *publish()* method on `com.calypso.tk.event.PSConnection`.
- Publish persistent events using the *saveAndPublish()* method on `com.calypso.tk.service.RemoteTrade`. This method saves and publishes an event or a list of events as a single transaction so events will only be published if saving to database was successful.

Sample Code

» Sample in `samples/cookbook/EventSubscriberPublisher.java`

3.1.2 Subscribing to Events

Once you have a PSConnection, you have to create a subscriber class that implements the `com.calypso.tk.event.PSSubscriber` interface. The client application will invoke the subscriber class to start subscription.

The subscriber class can do the following:

- Subscribe to non-persistent events using the *subscriber()* method on `com.calypso.tk.event.PSConnection`
- Subscribe to persistent events using one of the following methods on the remote services under `com.calypso.tk.service`:
  - *getEvents(String engineName, Vector eventClassNames, int max)* on RemoteTrade — Returns events processed by the given engine, and class names. The number of returned events will not exceed the given maximum.
  - *getEngineEvents(String engineName, Vector eventClassNames, int max)* on RemoteTrade — Returns compressed events processed by the given engine. The number of returned events will not exceed the given maximum. The returned events should be uncompressed using `com.calypso.tk.util.SerialUtil.bytes2object`.
  - *getEvents(String className, String where)* on RemoteAccess — Returns events of a given class name using the given SQL where clause on the `ps_event` table and the event table corresponding to the specified event class.

Sample Code

» Sample in `samples/cookbook/EventSubscriberPublisher.java`

3.2 How to Handle a Lost Connection to the Event Server

When writing a client program you may want to be notified and subsequently attempt to reconnect if the connection to the Event Server is lost. The following sample illustrates how to create a PSSubscriber that will automatically reconnect to the Event Server.

Create a timer in the *onDisconnect()* method to reconnect to the Event Server.

Sample Code

» Sample in `samples/cookbook/SmartSubscriber.java`

3.3 How to Create a Custom Event

New event types can be added to the system to accommodate the addition of custom objects. For example, when we add the EquityBasket in the Data Server extension example, we have to add a new event type PSEventEquityBasket to notify the system when an EquityBasket object is created or modified.

Overview of Steps

- Step 1 — Create an event class that extends PSEvent
- Step 2 — For persistent events only
- Step 3 — Register the new event class

3.3.1 Step 1 — Creating a Custom PSEvent

Create a class named `tk.event.PSEvent<event type>` that extends `com.calypso.tk.event.PSEvent`.

Note - Since event objects are serialized for communication, each event class needs to have its own unique serialVersionUID. This id can be obtained by running `serialVer.exe` and must be included in the class definition.

The following methods of PSEvent need to be implemented:

- *toString()* — Returns a canonical string representing the event object. The result includes the class name and identification number. However, the actual event subclasses will often want to redefine this method to produce a more descriptive description of the event.
- *getEventType()* — Returns the event class name of the event. However, the actual event subclasses will often have an event type that is more specific than the class name. Many subclasses redefine this method to return a more precise event type designation.

Sample Code

» Sample in `samples/cookbook/tk/event/PSEventEquityBasket.java`

3.3.2 Step 2 — For Persistent Events Only

Do the following for persistent events:

1. Create a persistent class named `tk.event.PSEvent<event type>SQL` that extends `com.calypso.tk.event.sql.PSEventSQL`.

2. Add the table for the event class to the database, along with any stored procedure.

The following methods of PSEventSQL need to be implemented:

- *saveInstance()* — Saves the given PSEvent to the database using the given connection. Returns true if successful or false if failed.

- *loadInstances()* — Returns a Vector of PSEvent objects from the database for the given the SQL where clause on the `ps_event` table, and connection. The number of events is limited to the given maximum.
- *deleteInstances()* — Deletes events with ids between the given minimum and maximum (inclusive). Returns true if delete is successful or false if failed.

  Note - The function returns true when no record is deleted because no records exist within the minimum and maximum.
- *purgeInstances()*

Tip

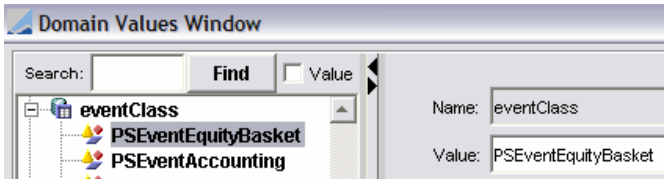
- It is generally good practice to keep an event class as small as possible for performance purposes. For data that can be obtained from the Data Server, it is recommended whenever possible, for the subscriber to obtain the data from the Data Server rather than having the publisher pass the object directly in an event class. This is particularly true if the data is a large object and is used only sparingly by subscribers.

Sample Code

- Sample for SQL class in `samples/cookbook/tk/event/sql/PSEventEquityBasketSQL.java`
- Sample for database scripts in `samples/cookbook/event_equity_basket.sql`

3.3.3 Step 3 — Registering the new Event Class

Add the new event class to the eventClass domain.



Section 4. Core

4.1 How to Create a Custom Daycount

Create a class named `tk.core.CustomDayCountCalculator` which implements the interface `com.calypso.tk.core.DayCountCalculator`.

This class will be invoked from `com.calypso.tk.core.DayCount` so that once it is compiled, the new Daycount will be made available in the system.

Note - The max length of a daycount is 9 characters.

Sample Code

- Sample in `calypsox/tk/core/CustomDayCountCalculator.java`

4.2 How to Create a Custom Tenor

Create a class named `tk.core.CustomTenorCalculator` which implements the interface `com.calypso.tk.core.TenorCalculator`.

This class will be invoked from `com.calypso.tk.core.Tenor` so that once it is compiled, the new Tenor will be made available in the system.

Sample Code

- Sample in `calypsox/tk/core/CustomTenorCalculator.java`

4.3 How to Create a Custom DateRule

Create a class named `tk.core.DateGenerator<custom_name>` that implements the interface `com.calypso.tk.core.DateGenerator`.

This class will be invoked from `com.calypso.tk.core.DateRule` so that once it is compiled, the new DateRule will be made available in the system.

4.4 How to Create a Custom Frequency

Create a class named `tk.core.CustomFrequencyCalculator` that implements the interface `com.calypso.tk.core.FrequencyCalculator`.

This class will be invoked from `com.calypso.tk.core.Frequency` so that once it is compiled, the new Frequency will be made available in the system.

Sample Code

- Sample in `calypsox/tk/core/CustomFrequencyCalculator.java`

4.5 How to Chain Exceptions

All calypso exceptions are derived from the class `com.calypso.tk.core.CalypsoException`. The exceptions can be chained using *setNext()* and *getNext()* to allow the system to throw multiple exceptions.

In all SQL transactions under `tk`, the code catches all Throwable rather that just Exception.

4.6 How to Create a Comparator Class for Sorting Objects

Create a class named `tk.util.<name>Comparator` that implements `java.util.Comparator`.

You can access the comparator class using *ComparatorFactory.getCustomComparator("<name>")*.

You can create a custom comparator class to order aggregation groups in ScenarioAnalysisViewer.

Section 5. Engines

An engine is an application which responds automatically to a certain type of event occurring on the system.

5.1 How to Create a Custom Engine

An engine is implemented by sub-classing the Engine abstract base class. The Engine class encapsulates the objects and core features that are commonly required by a Calypso engine such as multi-threaded handling of events and automatic handling of core Calypso event types.

Overview of Steps

- Step 1 — Create a custom Engine class
- Step 2 — Register the new engine
- Step 3 — For subscribing to persistent events only
- Step 4 — Start the new engine

5.1.1 Step 1 — Creating a Custom Engine Class

Create a class that extends `com.calypso.engine.Engine`. There is no restriction regarding the name of the engine or the location of the class.

The constructor for the Engine class is as follows:

- *Engine(DSConnection dsCon, String hostname, int port)*
  - `dsCon` — a live connection to the Data Server.
  - `hostname` — the String name of the machine where the Event Server is running. You can find this name by calling *com.calypso.tk.core.Defaults.getESHost()* or *com.calypso.tk.service.RemoteAccess.getEventServerHost()*.
  - `port` — the int port number of the Event Server. You can find this number by calling com.calypso.tk.core.Defaults.getESPort() or com.calypso.tk.service.RemoteAccess.getEventServerPort().

The following methods of Engine need to be implemented:

- *getEngineName()* — Returns the engine name. The engine name must be unique from other engines in the system. Engine names are specified in the engineName domain.
- *getClasses()* — Returns all event classes that the engine subscribes to (both persistent and non-persistent events).

  Persistent events that an engine subscribes to are specified in EventConfig. See below.
- *process(PSEvent event)* — Returns "true" if the event was successfully processed, or "false" otherwise. The *process()* method must call *RemoteTrade.eventProcessed()* to notify the Event Server if the event being processed is of the type that the engine subscribes to.

Sample Code

» Sample in `samples/SampleEngine.java`

Note - In order to run this sample program, you need to register the PSEventSample and PSEventSampleB classes with the system by adding the event class names to the eventClass domain. This is in addition to registering the engine as described below.

Tips

- **Managing subscriptions at runtime** — Some applications might need to add or remove subscription to events upon user input at runtime. To add a subscription, you would need to get the PSConnection object from the engine and call *PSConnection.subscribe()*. To remove a subscription, call *PSConnection.unsubscribe()*.

```
// Add subscription
engine.getPSConnection().subscribe(new PSEventTime().getClass().getName());
// Remove subscription
engine.getPSConnection().unsubscribe(new PSEventTime().getClass().getName());
```

- **Add publishing to an engine** — An engine will often publish events as well as subscribe to events. To publish within an engine you would simply add the following to the engine *getPSConnection().publish()*.

- **Event Filtering** — A mechanism is provided for filtering the events which will be received by an engine for processing. You may wish to implement event filtering for performance reasons, by decreasing the number of unnecessary events sent to an engine. Or you may wish to divide the processing workload across multiple engines, by segregating the events using a filtering scheme.

  Event filters are registered in the eventFilter domain and configured in the Filters panel under Main Entry > Configuration > System > Event Configuration.

  Create a class named `tk.event.<event_filter_name>` that implements `com.calypso.tk.event.EventFilter`. This class will be invoked from `com.calypso.tk.event.EventConfig`.

  Samples

  - An example of a performance filter may be found in `com/calypso/tk/event/VerifiedEventFilter.java`. This filter returns only PSEventTrade objects for which the trade status is not one of NONE, PRICING or PENDING. This filter is used by the Transfer Engine in Calypso's standard setup. Since we wish only to generate transfers for trades which have reached the VERIFIED status, we filter out PSEventTrade objects for trades which have not reached the VERIFIED status.

  - An example of filtering to segregate workload may be found in `calypsox/tk/event/BookOnlyEventFilter.java` and `calypsox/tk/event/NotBookOnlyEventFilter.java`. The first filter returns only PSEventTrade for trades in book TRADINGC from Calypso's standard setup. The second filter returns only PSEventTrade for trades not in book TRADINGC. The first filter may then be used by `calypsox/engine/payment/SampleTransferEngine1.java` and the second by `calypsox/engine/payment/SampleTransferEngine2.java`.

  - See `calypsox/engine/payment/sampleTransfer_SYBASE.sql` for information on setting up the engines with their respective filters.

  To customize the user's event filter definition, create a class named `tk.event.ClientEventFilterDescription` that implements `com.calypso.tk.event.CustomClientEventFilter`.

  Sample in `calypsox/tk/event/ClientEventFilterDescription.java`.

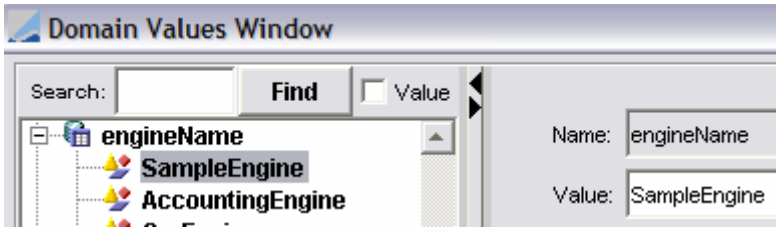
- **Setting engine parameters** — You can define parameters in your engine that can be saved and viewed in the Administrator Window > Engine Thread tab. Use the following methods on `com.calypso.tk.service.RemoteAccess` to load and save engine parameters.

  - *saveEngineParams(Hashtable h)*
  - *getEngineParams()*
  - *getEngineParam(String engine, String param)*

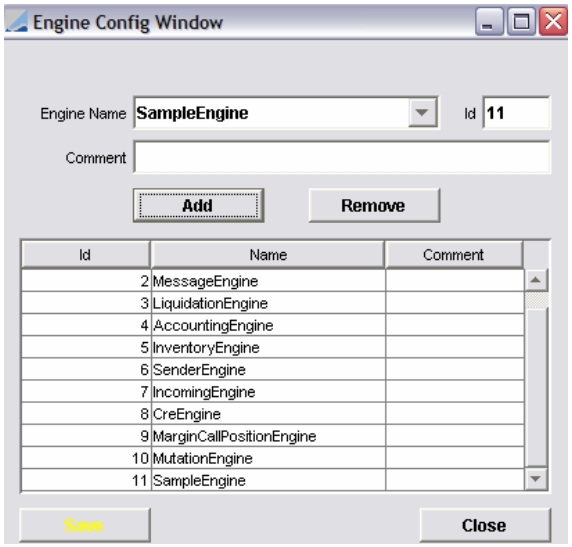
5.1.2 Step 2 — Registering the new Engine

Do the following for registering the new engine:

1. Add the engine name to the engineName domain.



2. Add configuration for the new engine using Main Entry > Configuration > System > Engine Configuration.

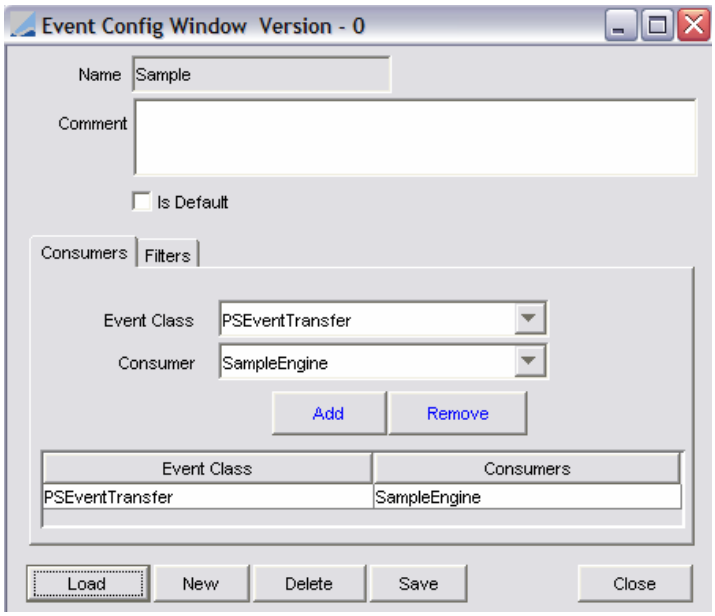


Note - You should use the next available engine id as to not impact the performance (choosing an engine id of 100 is not recommended if the next available engine id is 11).

If your engine id is used by Calypso in a subsequent release, you only need to make sure that all events have been processed, and change the engine id to the next available engine id again.

5.1.3 Step 3 — For Subscribing to Persistent Events Only

Specify the persistent events that the engine subscribes to using Main Entry > Configuration > System > Event Configuration.



Persistent events will be returned by *Engine.getPersistentClasses()*.

Don't forget to specify event filters if any.

5.1.4 Step 4 — Starting the new Engine

In the client application, create a connection to the Data Server, instantiate the engine class and call *start(Boolean doBatch)* on the engine. If doBatch is set to true, the engine will call *RemoteTrade.getEngineEvents()* in a separate thread to load and process any outstanding persistent events.

When *start()* is called on the engine, the engine starts a connection to the Event Server. The Event Server, based on the event types returned from the method *getPersistentClasses()*, sends the application any events that the persistent engine subscribes to and has missed since the last session. Upon receiving an event, the engine automatically creates a new thread and calls the method *process()*. If the event received is of a type that the engine subscribes to, the engine notifies the Event Server the completion of processing the event by calling *RemoteTrade.eventProcessed()*.

You can also use the sample engine that subscribes to PSEventTransfer events in order to process payments. The subscription is established in the database. Refer to the script `samples/sql/sampleEngine.sql` to see how this is done.

5.2 How to Customize the Transfer Engine

The Transfer engine generates transfers using the following process based on a BOProductHandler for a given product and the Transfer rules if settlement instructions have been set up.

- The BOProductHandler will create all the transfers relating to a Trade by using the cash flows and applying the settlement instructions.

  If the Transfer Engine is processing the Trade for the first time, all transfers are generated from the beginning of the trade. If the transfer engine has already processed the trade in a previous session, it filters the Transfers to the ones having a settle date after or on the current day's date and which do not have a CANCELED status. If the event is a PSEventProcessTrade, the limit date is specified in the event for back-value payment regeneration. You can create a custom date filter, and you can customize how to set the dates on the transfers.

- The Transfer Engine then compares the generated transfers to the existing ones and matches them based on the following list of criteria. If a transfer does not match, it is considered unmatched.

- The netting function will create netted transfers if netting is required by the transfers. The Netting Type field on the Transfer indicates if netting is required and what type of netting should occur. You can customize the netting selector.

5.2.1 Creating a Custom BOProductHandler

The base class for producing trade cashflows and transfers with settlement instructions is `com.calypso.tk.bo.BOProductHandler`. Each product will usually have its own BOProductHandler. A BOProduct<product_type>Handler can itself extend or override another BOProduct<product_type>Handler.

Create a class named `tk.bo.BO<product_type>Handler` or `tk.bo.BO<product_family>Handler` which extends `com.calypso.tk.bo.BOProductHandler`.

A BOProductHandler may define the following methods: *generateTransferRules*, *generateTransfers*, *exercise*, and *addSecurityFlows*.

This class will be invoked from `com.calypso.tk.bo.BOProductHandler`.

Sample Code

- Samples in `calypsox/tk/bo/BODEMO_P1Handler.java`, `calypsox/tk/bo/BODEMO_P2Handler.java`, `calypsox/tk/bo/BOIROptionHandler.java`, and `calypsox/tk/bo/BOStraddleHandler.java`
- A custom `calypsox/tk/bo/BORepoHandler.java` extends `com.calypso.tk.bo.BORepoHandler`. In case a coupon occurs during the life of a repo, no messages or postings are generated for the transfers related to this coupon.

5.2.2 Creating a Custom Interest Dispatch Process

It is possible to customize the behavior of the interest dispatch process for repos.

Create a class named `tk.bo.BORepo<dispatch_method>DispatchInterestHandler` that implements `com.calypso.tk.bo.BORepoDispatchInterestHandler`.

BORepoDispatchInterestHandler has two methods:

- *dispatchInterest()* called from *addSecurityFlow()*, allows creating as many interest flows as necessary.
- *updateInteretTransfer()* called from *preProcessDAPTransfers()*, allows linking transfers created because of the new interest flows to the security and principal transfers of each collateral.

Then register the dispatch method with the "Repo.DispatchInterestMethod" domain.

This class will be invoked from `com.calypso.apps.tk.bo.BORepoHandler`.

5.2.3 Creating a Custom Date Filter

The default behavior will generate the Product transfers up to the next event date if the XFER_NEXT_EVENT parameter is set to "true".

Create a class named `tk.product.<product_type>ProductNextEventDate` that implements the interface `com.calypso.tk.product.ProductNextEventDate`.

This class will be invoked from `com.calypso.tk.product.ProductNextEventDateUtil` which is used by BOProductHandler.

5.2.4 Creating a Custom Date Selector

Create a class named `tk.bo.CustomBOTransferDateSelector` which implements the interface `com.calypso.tk.bo.BOTransferDateSelector`.

This class will be invoked from com.calypso.tk.bo.BOProductHandler to set the dates on the transfers.

Sample Code

- Sample in calypsox/tk/bo/SampleCustomBOTransferDateSelector.java

5.2.5 Creating a Custom Transfer Matching Mechanism

This interface is used to determine whether a Transfer needs to be canceled, updated or created. For instance, it allows the user to disable some criteria during the matching of two transfers.

Create a class named engine.payment.<product_type>TransferMatching or engine.payment.DefaultTransferMatching which implements the interface com.calypso.engine.payment.TransferMatching.

This class will be invoked from com.calypso.engine.payment.TransferMatchingUtil, which is used by TransferEngine to select the transfer matching mechanism.

The default transfer matching mechanism is based on the CHECK_PAST_SDI_VERSION property. If CHECK_PAST_SDI_VERSION is set to "false" and the SDI version number has changed, transfers will not be regenerated. If CHECK_PAST_SDI_VERSION is set to "true" and the SDI version number has changed, transfers will be regenerated.

Sample Code

- Sample in calypsox/engine/payment/SampleDefaultTransferMatching.java

5.2.6 Creating a Custom Netting Method Selector

This automatically sets the Netting Type on the Transfer Rules and lets you override the Netting Method default value.

Create a class named tk.bo.<product_type>NettingSelector which implements the interface com.calypso.tk.bo.NettingSelector.

This class will be invoked from com.calypso.tk.bo.NettingSelectorUtil to set the netting method.

Sample Code

- Samples in calypsox/tk/bo/SimpleMMNettingSelector.java and calypsox/tk/bo/SampleCustomBondNettingSelector.java

5.2.7 Creating a Custom Persistence Routine for Transfer Attributes

Create a class named tk.bo.sql.CustomTransferAttributeSQL that implements the interface com.calypso.tk.bo.sql.TransferAttributeSQL.

This class will be invoked from com.calypso.tk.bo.sql.BOTransferSQL when saving transfers.

TransferAttributeSQL includes methods for archiving custom attributes that need to be implemented. To fully support archiving and restoring of custom attributes, you must define new history tables for any custom attribute tables you have added to the database.

5.3 How to Customize the Message Engine

The Message engine generates messages for various events as applicable (Trade, Transfer, etc), using the following process:

- The Message engine identifies the roles to which sending the message using TradeRoleFinder. Once, the roles are determined, receiver contact information can be retrieved. A message is generated for each receiver using a BOMessageHandler provided MessageSelector confirms the message needs to be

generated. The behavior of TradeRoleFinder and MessageSelector can be customized. You can create a custom BOMessageHandler for a given product.

- BOMessageHandler builds the messages based on the Message setup rules. It uses FormatterUtil to select a type of Formatter (SWIFT, HTML, etc.) and a template. It uses MessageFormatterUtil to select a MessageFormatter for populating the template. The MessageFormatter is selected based on the type of Formatter and the product. You can create a custom type of Formatter, a custom template selector, a custom MessageFormatter selector, and a custom MessageFormatter.

5.3.1 Creating a Custom Role Finder

For example, you want to retrieve the possible receivers of a message based on the Legal Entity role. Write this class to retrieve the Legal Entities for a given Role defined in a Trade, a Product or a Transfer.

Create a class named `tk.bo.<product_type>RoleFinder` or `tk.bo.<product_family>RoleFinder` which extends the class `com.calypso.tk.bo.TradeRoleFinder`.

This class will be invoked from `com.calypso.tk.bo.TradeRoleFinder`.

Sample Code

» Samples in `com/calypso/tk/bo/TransferAgentRoleFinder.java` and `calypsox/tk/bo/RepoRoleFinder.java`

5.3.2 Creating a Custom Message Selector

For instance, the SampleCustomMessageSelector returns false when there is no SWIFT address for the Receiver Contact, causing a message to not be generated by the Message engine.

Create a class named `engine.advice.CustomMessageSelector` which implements the interface `com.calypso.engine.advice.MessageSelector`.

This class will be invoked from `com.calypso.engine.advice.MessageEngine` to establish if a message needs to be generated.

Sample Code

» Samples in `calypsox/engine/advice/SampleCustomMessageSelector.java`, `calypsox/engine/advice/AnotherCustomMessageSelector.java`, and `calypsox/engine/advice/YetAnotherCustomMessageSelector.java`

5.3.3 Creating a Custom Message Handler

Create a class named `tk.bo.BO<product_type>MessageHandler` or `tk.bo.BO<product_family>MessageHandler` which extends `com.calypso.tk.bo.BOMessageHandler`.

Note - For message types that do not have a product type, like STATEMENT, you can create a class named `tk.bo.BON/AMessageHandler`.

The following methods in the BOMessageHandler may be redefined:

- *generateBOMessages()* — Defines what messages should be produced. For example, for an FX Swap confirmation by Swift, it would create two messages, one for each leg.

- *setSpecialMessageEnvironment()* — The Message engine calls this function whenever an existing message has been found. It allows you to set a link between each message and decide what should be done on the previous existing message. Typically this function sets the keyword AMEND, CANCEL or NEW.

- *filterMessages()* — Returns the existing message matching exactly the new one which should be generated. For example, in case you have already produced a Bond Confirmation, it will return the existing Bond Confirmation already generated for the Trade.

- *isMessageRequired()* — Allows you to decide if the new message should be created. In this function you have access to the previous message generated. You can therefore perform any type of comparison required.

This class will be invoked from `com.calypso.tk.bo.BOMessageHandler`.

5.3.4 Creating a Custom Type of Formatter

Out-of-the-box, the HTML, Text, SWIFT and XML types are supported. For example, it might be necessary to support the FIX format and hence a FIX generator class.

Create a class named `tk.bo.<format_type>Formatter` which implements the interface `com.calypso.tk.bo.Formatter`. Implement the methods *generate()* to generates an advice document, and *display()* to displays the advice document in the Task Station.

This class will be invoked from `com.calypso.tk.bo.FormatterUtil`.

5.3.5 Creating a Custom Template Selector

Create a class named `tk.bo.<product_type>TemplateSelector` which implements the interface `com.calypso.tk.bo.TemplateSelector`.

This class will be invoked from `com.calypso.tk.bo.FormatterUtil` to choose a template selector.

Sample Code

» Sample in `calypsox/tk/bo/SampleSwapTemplateSelector.java`

5.3.6 Creating a Custom Message Formatter Selector

Create a class named `tk.bo.<message_type>MFSelector` which implements the interface `com.calypso.tk.bo.MFSelector`.

This class will be invoked from `com.calypso.tk.bo.MessageFormatterUtil` to choose a MessageFormatter selector.

5.3.7 Creating a Custom Message Formatter

A Message Formatter is responsible for extracting the information from a trade and matching the appropriate keywords in the template. See How to Create an HTML Template for information on how to create an HTML template.

Create a class named `tk.bo.<message_type><product_type>MessageFormatter`, `tk.bo.<product_type>MessageFormatter` or `tk.bo.CustomMessageFormatter` which extends `tk.bo.MessageFormatter`.

In your MessageFormatter, you will create a *parse<keyword_name>()* method for each keyword you wish to add. For example *parseRATE_INDEX()*.

Implement each parse method to take the following arguments and return the keyword value for a given situation, as defined by the passed arguments.

- `message` — the message which will use the returned keyword value;
- `trade` — the trade with which the advice is associated;
- `sender` — the contact person sending the advice (LEContact);
- `rec` — the contact person receiving the advice (LEContact);

- `transferRules` — a Vector of TradeTransferRule objects which provide the general definition of any payments associated with the advice;
- `transfer` — the payment, if any, associated with the advice;
- `dsCon` — a connection to the Data Server.

Note - In your parse method, you have the possibility to check if the custom keyword is being evaluated within an IF statement using *FormatterParser.isConditionalEvaluation()*, which will return a Boolean.

A MessageFormatter class will be invoked from `com.calypso.tk.bo.MessageFormatterUtil`.

CustomMessageFormatter will be invoked from `com.calypso.tk.bo.MessageFormatter`.

Sample Code

» Samples in `calypsox/tk/bo/RATE_RESETSwapMessageFormatter.java`, `calypsox/tk/bo/StructuredProductMessageFormatter.java`, `calypsox/tk/bo/SwapMessageFormatter.java` and `calypsox/tk/bo/XLSMessageFormatter.java`

5.3.8 Creating a Custom Persistence Routine for Message Attributes

Create a class named `tk.bo.sql.CustomMessageAttributeSQL` that implements the interface `com.calypso.tk.bo.sql.MessageAttributeSQL`.

This class will be invoked from `com.calypso.tk.bo.sql.BOMessageSQL` when saving messages.

MessageAttributeSQL includes methods for archiving custom attributes that need to be implemented. To fully support archiving and restoring of custom attributes, you must define new history tables for any custom attribute tables you have added to the database.

5.3.9 Creating a Custom XML Generator

Create a class named `tk.bo.xml.<template_name>XMLGenerator` or `tk.bo.xml.DefaultXMLGenerator` which implements the interface `com.calypso.tk.bo.xml.XMLGenerator`.

This class will be invoked from `com.calypso.tk.bo.xml.XMLUtil`.

Sample Code

» Sample in `calypsox/apps/util/XMLGenerator.java`

5.3.10 Creating Multiple BOMessages per Message Type

In a class named `tk.product.<product_type>` implement the interface `com.calypso.tk.product.MultiMessageProduct`.

For example, an FX Swap needs 2 trade confirmations when you verify the trade; one confirm for the near leg and another confirm for the far leg.

5.3.11 How to Customize a Statement Message

The interface AccountStatementInterface in order to provide flexibility in the population of the fields: *fillMessageReference(MessageArray message, AccountStatement statement, DSConnection dsCon)*. It is used by the Message engine to set the Message Reference on the Statement messages.

Create a class named `tk.bo.swift.AccountStatementCustomizer` that implements AccountStatementInterface to customize the statements.

5.4 How to Customize SWIFT Messages

SWIFT messages can be generated using a SwiftGenerator for each type of message, or from an XML template using SWIFTFormatter. Based on the template name, the message engine first tries to instantiate a SWIFTFormatter. If no SWIFTFormatter is available, the message engine uses a SwiftGenerator.

5.4.1 Using SwiftGenerator

Creating a Custom SwiftGenerator

Create a class named `tk.bo.swift.<template_name>SwiftGenerator` which implements the `com.calypso.tk.bo.swift.SwiftGenerator` interface.

For example, if you are creating a SWIFT message for FX Swap payments then the name of the class might be FXSwapPaymentSwiftGenerator. You will have to add the template name in the domain values. Add the domain FXSwapPayment to the domain "SWIFT.Templates" so that the template will be selectable in the message setup window.

This class will be invoked from `com.calypso.tk.bo.SWIFTFormatterUtil`.

Sample Code

- Samples in `calypsox/tk/bo/swift/FRAConfirmSwiftGenerator.java`, `calypsox/tk/bo/swift/FXPaymentSwiftGenerator.java`, `calypsox/tk/bo/swift/FXReceiptSwiftGenerator.java`, `calypsox/tk/bo/swift/FXSwapPaymentSwiftGenerator.java`, and `calypsox/tk/bo/swift/FXSwapReceiptSwiftGenerator.java`

5.4.2 Using SWIFTFormatter

Creating a Custom SWIFTFormatter

This allows creating a custom SWIFTFormatter that does not use SwiftGenerator. The XML templates should be placed in `templates/swift`. See SWIFTFormatter Javadoc for details.

Create a class named `tk.bo.swift.<name>SWIFTFormatter` that extends `com.calypso.tk.bo.swift.SWIFTFormatter`.

This class will be invoked from `com.calypso.tk.bo.SWIFTFormatterUtil`.

Creating a Custom Iterator to Populate Repeated Information

Create a class named `tk.bo.swift.<name>Iterator`, `tk.bo.swift.<name>`, or `<name>` that implements the interface `java.util.Iterator`.

The SWIFTFormatter can access the iterator count and the iterator object, and behave as applicable. See SWIFTFormatter Javadoc for details.

Sample Code

- Sample in `calypsox/tk/bo/swift/TestIterator.java`

Creating a Custom Header Block

Create a class named `tk.bo.swift.SwiftTextCustomizer` which implements the interface `com.calypso.tk.bo.swift.SwiftTextInterface`.

Sample Code

- Sample in `calypsox/tk/bo/swift/SwiftTextCustomizer.java`

5.4.3 Applying Custom Validation to SWIFT Messages

Create a class named `tk.bo.swift.CustomSwiftMessageValidator` which implements the interface `com.calypso.tk.bo.swift.SwiftMessageValidator`.

This class will be invoked from `com.calypso.tk.bo.swift.SwiftMessage` before the message is saved.

Sample Code

- Sample in `calypsox/tk/bo/swift/SampleCustomSwiftMessageValidator.java`

5.4.4 Creating a Custom EntityInfo for SWIFT Messages

Create a class named `tk.bo.swift.CustomEntityInfo` that implements `EntityInfo`.

This class will be invoked from `com.calypso.tk.bo.swift.SwiftUtil`.

5.5 How to Customize the Sender Engine

The Sender engine uses DocumentSender objects to physically transmit message documents to a given address method or gateway. Out-of-the-box, Calypso provides the EMAILDocumentSender to send documents via e-mail. Address methods are stored in the addressMethod domain.

5.5.1 Creating a Custom Document Sender

Create a class named `tk.bo.document.<address_method>DocumentSender`, `tk.bo.document.Gateway<gateway>DocumentSender`, or `tk.bo.document.<address_method>Gateway<gateway>DocumentSender` that implements the `com.calypso.tk.bo.document.DocumentSender` interface.

This class will be invoked by `com.calypso.tk.bo.document.DocumentSenderUtil`.

In your DocumentSender, you will create the *send()* method to send the message. Generally the *send()* method will initiate the physical production of the document via some output mechanism such as a printer or email utility. The *send()* method should return a Boolean of true if the send succeeds, or false otherwise. You must also define the *isOnline()* method. The Sender engine will query this method to make sure the sender gateway system is online.

DocumentSender also has the ability to process a PSEventMessage event in addition to sending an Advice document. The following parameters of the *send()* method should be used as described below:

- `saved` — `saved[0]` should be true to indicate if the document was saved and the event processed
- `engineName` — should be `null` to indicate that the Sender engine does not need to process the event

Sample Code

- Samples in `calypsox/tk/bo/document/GatewayPRINTERDocumentSender.java` and `calypsox/tk/bo/document/SWIFTDocumentSender.java`

5.6 How to Customize the Accounting Engine

The Accounting engine generates postings for various events as applicable (Trade, Valuation, Liquidation, etc), using the following process:

- The Accounting engine selects what accounting rule to apply using a mapping mechanism between the events it subscribes to and the accounting rules configured in the system. It builds a list of requested accounting events based on the selected accounting rule. The mapping mechanism can be customized.

- For each accounting event, the Accounting engine calls a generic AccountingHandler to specify how to generate the corresponding posting. AccountingHandler can call a specific AccountingHandler for a given product type, or a specific AccountingEventHandler for a given type of accounting event. The Accounting engine also allows adding custom attributes to the generated postings.

- Once all the postings have been created, a matching process occurs to compare a set of criteria on the new postings and the old postings, and to generate reverse postings as applicable. The matching mechanism can be customized.

5.6.1 Creating a Custom Mapping Mechanism to an Accounting Rule

Create a class named `engine.accounting.CustomAccountingRuleSelector` which implements the interface `com.calypso.engine.accounting.AccountingRuleSelector`.

This class will be invoked from `com.calypso.engine.accounting.AccountingEngine` to select an accounting rule.

Sample Code

» Sample in `calypsox/engine/accounting/SampleCustomAccountingRuleSelector.java`

5.6.2 Creating a Custom Accounting Handler

Create a class named `tk.bo.accounting.<product_type>AccountingHandler` or `tk.bo.accounting.<product_family>AccountingHandler` which extends `com.calypso.tk.bo.accounting.AccountingHandler`.

The AccountingHandler should have a *get<accounting event type>()* method for each accounting event type that it will calculate. Accounting event types are listed in the accEventType domain. The set of accounting event types that your system will calculate is established in the AccountingEventConfig objects for a given AccountingRule. For example, *getCOT()* calculates a COT accounting event.

This class will be invoked from `com.calypso.tk.bo.accounting.AccountingHandler` to generate a posting for a given product type of family type.

Sample Code

» Samples in `calypsox/tk/bo/accounting/FXForwardTakeUpAccountingHandler.java` and `calypsox/tk/bo/accounting/IROptionAccountingHandler.java`

5.6.3 Creating a Custom Event Accounting Handler

Create a class named `tk.bo.accounting.<event_type>AccountingHandler` which implements the interface `com.calypso.tk.bo.accounting.EventAccountingHandler`.

This class will be invoked from `com.calypso.tk.bo.accounting.AccountingHandler` to generate a posting for a given type of accounting event.

Sample Code

» Samples in `calypsox/tk/bo/accounting/EXT_EVENT_TYPEAccountingHandler.java` and `calypsox/tk/bo/accounting/EXT_MTM_FULLAccountingHandler.java`

5.6.4 Creating a Custom Posting Description

Create a class named `engine.accounting.CustomFillPostingDescription` which implements the interface `com.calypso.engine.accounting.FillPostingDescription`. You can implement *fillDescription()* for adding attributes and *fillPostingDates()* for customizing the dates set on the posting: booking date and effective date.

This class will be invoked from `com.calypso.engine.accounting.AccountingEngine` for customizing the content of the postings.

Sample Code

- Sample in calypsox/engine/accounting/SampleCustomFillPostingDescription.java
- For example, this API is used to fulfill the following request: in order to freeze the image of the postings during the EOD process, the posting status is changed from NEW to EOD_PROCESSING in *fillDescription()*. Hence if any cancellation occurs during the EOD process, the EOD_PROCESSING posting will be reversed and a NEW posting will be created. Otherwise the original posting would just move to status DELETE.

5.6.5 Creating a Custom Accounting Matching Mechanism

Create a class named engine.accounting.<product_type>AccountingMatching or engine.accounting.DefaultAccountingMatching which implements the interface com.calypso.engine.accounting.AccountingMatching.

This class will be invoked from com.calypso.engine.accounting.AccountingMatchingUtil when matching old postings and new postings for generating reverse postings if applicable.

Sample Code

- Sample in calypsox/engine/accounting/SampleSwapAccountingMatching.java

5.6.6 Creating a Custom Account Keyword for Automatic Accounts

Create a class named tk.bo.accounting.keyword.<keyword>AccountKeyword which implements the interface com.calypso.tk.bo.accounting.keyword.AccountKeyword.

This class will be invoked from com.calypso.tk.bo.accounting.keyword.KeywordUtil.

Then register the <keyword> in the attributeType domain. For example for calypsox/tk/bo/accounting/keyword/IBANAccountKeyword.java, you need to register IBAN in the attributeType domain.

Sample Code

- Samples in calypsox/tk/bo/accounting/keyword/IBANAccountKeyword.java, calypsox/tk/bo/accounting/keyword/InitialMaturityAccountKeyword.java, calypsox/tk/bo/accounting/keyword/MatTenorAccountKeyword.java, calypsox/tk/bo/accounting/keyword/MethodAccountKeyword.java, calypsox/tk/bo/accounting/keyword/OnTimeAccountKeyword.java, and calypsox/tk/bo/accounting/keyword/RIBAccountKeyword.java

5.6.7 Applying Custom Validation to Accounting Rules

Create a class named apps.refdata.CustomAccRuleValidator which implements the interface com.calypso.apps.refdata.AccRuleValidator.

This class will be invoked from com.calypso.apps.refdata.AccountingRuleFrame prior to saving an accounting rule.

5.6.8 Applying Custom Validation to an Account

Create a class named apps.refdata.CustomAccountValidator that implements com.calypso.apps.refdata.AccountValidator.

This class will be invoked from com.calypso.apps.refdata.AccountFrame.

Sample Code

- Sample in calypsox/apps/refdata/CustomAccountValidator.java

5.6.9 Creating a Custom Closing Account Name

Create a class named `tk.bo.accounting.CustomClosingAccountName` which implements the interface `com.calypso.tk.bo.accounting.ClosingAccountName`.

This class will be invoked from `com.calypso.tk.bo.BalanceUtil` when assigning a closing account.

Sample Code

- Sample in `calypsox/tk/bo/accounting/SampleCustomClosingAccountName.java`

5.6.10 Creating a Custom External Name for Automatic Accounts

Create a class named `tk.bo.accounting.keyword.CustomAccountExternalName` that implements `com.calypso.tk.bo.accounting.keyword.AccountExternalName`.

This class will be invoked from `com.calypso.tk.bo.accounting.keyword.KeywordUtil`.

Sample Code

- Sample in `calypsox/tk/bo/accounting/keyword/CustomAccountExternalName.java`

5.7 How to Customize the Position Engine

5.7.1 Creating a Custom Liquidation Method

Create a class named `engine.position.Liquidation<liquidation_method>` which extends the class `com.calypso.engine.position.Liquidation`. Out-of-the-box liquidation methods are stored in the liquidationMethod domain.

This class will be invoked from `com.calypso.engine.position.LiquidationUtil`.

5.7.2 Creating a Custom Sort Method

Create a class named `tk.mo.Comparator<sort_method>` that implements `java.util.Comparator`. Out-of-the-box sort methods are stored in the sortMethod domain.

This class will be invoked from `com.calypso.engine.position.LiquidationUtil` for sorting open positions.

5.7.3 Creating a Custom Routine for Computing the Liquidation Date

Create a class named `tk.mo.LiquidationDateCalculator` that implements the interface `com.calypso.tk.mo.LiquidationInfoCalculator`.

This class will be invoked from `com.calypso.tk.mo.LiquidationInfo`.

Sample Code

- Sample in `calypsox/tk/mo/LiquidationDateCalculator.java`

5.8 How to Customize the Inventory Engine

5.8.1 Creating a Custom Inventory Position Selector

For example, you want to customize the list of Positions classes handled by the Inventory Engine: INTERNAL, CLIENT, EXTERNAL.

Create a class named `engine.inventory.InventoryPositionSelector` which implements the interface `com.calypso.engine.inventory.PositionSelector`.

This class will be invoked from `com.calypso.engine.inventory.InventoryEngine`.

Sample Code

- Sample in `calypsox/engine/inventory/SampleInventoryPositionSelector.java`

5.9 How to Customize the CRE Engine

The CRE engine generates CREs (accounting events). The CRE engine calls a generic CreHandler to specify how to generate a CRE. CreHandler can call a specific CreHandler for a given product type, or a specific CreEventHandler for a given accounting event. CreHandler also allows adding custom attributes to the generated CREs.

5.9.1 Creating a Custom CRE Handler

Create a class named `tk.bo.accounting.<product_type>CreHandler` or `tk.bo.accounting.<product_family>CreHandler` which extends `com.calypso.tk.bo.accounting.CreHandler`.

Implement a *get<accounting event type>()* method for each accounting event. For example, *getCOT()* for the COT accounting event.

This class will be invoked from `com.calypso.tk.bo.accounting.CreHandler` to generate a CRE for a given product type of family type.

5.9.2 Creating a Custom Event CRE Handler

Create a class named `tk.bo.accounting.<accouting_event_type>CreHandler` which implements the interface `com.calypso.tk.bo.accounting.EventCreHandler`.

This class will be invoked from `com.calypso.tk.bo.accounting.CreHandler` to generate a CRE for a given type of accounting event.

5.9.3 Creating a Custom CRE Description

Create a class named `com.calypso.tk.bo.accounting.CustomFillCreDescription` that implements `com.calypso.tk.bo.accounting.FillCreDescription`.

This class will be invoked from `com.calypso.tk.bo.accounting.CreHandler` for adding custom attributes to the generated CREs.

Sample Code

- Sample in `calypsox/tk/bo/accounting/SampleCustomFillCreDescription.java`

5.9.4 Creating a Custom Persistence Routine for CRE Attributes

Create a class named `tk.bo.sql.CustomCreAttributeSQL` that implements `tk.bo.sql.CreAttributeSQL`.

CreAttributeSQL includes methods for archiving custom attributes that need to be implemented. To fully support archiving and restoring of custom attributes, you must define new history tables for any custom attribute tables you have added to the database.

This class will be invoked from `com.calypso.tk.bo.sql.BOCreSQL` when saving CREs.

5.10 How to Customize the CRE Sender Engine

The CRE Sender engine sends the CREs generated by the CRE engine. During the send, the status of a CRE is updated (to SENT, RE_SENT or DELETED), and the CreSenderFormater API is called to produce the output of a CRE. You have to implement CreSenderFormater. Note that the scheduled task CRE_SENDER also calls CreSenderFormater for formatting CREs.

5.10.1 Creating a Custom CRE Formatter

Create a class named `engine.accounting.CreSenderFormaterImpl` that implements `com.calypso.engine.accounting.CreSenderFormater`.

This class will be invoked from `com.calypso.tk.bo.sql.BOCreSQL` when saving CREs.

Section 6. Limits

6.1 How to Create Custom Limit Types

Create a class named `tk.limit.<LimitType>Limit` that extends the abstract base class `com.calypso.tk.limit.BaseLimit`.

Register custom limit types in the limitType domain.

6.2 How to Exclude Trades from Limit Checking

You can now specify a rule to exclude trades from the limit check other than using a trade filter, or the limit exclude keyword.

The exclusion rule is implemented using the EXCLUSION type of limit. It does not compute limit usages but only determines whether a given trade should be included or not. Using the EXCLUSION type of limit does not replace the trade filter or limit exclude keyword, but represents a third more custom way to exclude a trade.

To specify an EXCLUSION type of limit, implement a class called `tk.limit.<product_type>EXCLUSIONLimit`, or `tk.limitEXCLUSIONLimit` that implements `com.calypso.tk.limit.BaseLimit`.

If no such class is implemented, no trade will be excluded (other than trades excluded by the trade filter or using the limit exclude keyword).

Section 7. Message Documents

Messages in Calypso are converted into documents prior to being physically sent out of the system. These documents may also be edited and stored in the database prior to being sent. Out-of-the-box, the following document formats are supported: HTML, text, XML and SWIFT.

7.1 How to Create an HTML Template

Templates are supported for the HTML format. Calypso provides a standard set of document templates. You may wish to modify them or to create your own. Templates are associated with messages using Main Entry > Configuration > Messages & Matching > Message Set-up Configuration.

HTML templates contain the text and the format of message documents such as confirmations, payment or receipt advices or any other message document generated based upon the Message Setup (refer to the *Calypso Messages User Guide* for details). Any information that is required from the trade, the message or the transfer is marked as keywords in the template. The MessageFormatter will extract the information from the trade and populate the template keywords. Conditional processing is also supported in order to allow more flexibility in structuring the templates. For example, common header and footer information may be kept in their own files and included in other templates, or sub documents may be included based upon conditions.

Calypso templates are located under `resources.com.calypso.templates`. Custom templates should be located under `resources.<custom package name>.templates`. A list of the keywords available for building your own message templates can be seen in Main Entry > Help > Message Template Keywords.

Keywords have the format `|keyword name|`. This section focuses on conditional keywords with the format: `<calypso>conditional keyword name</calypso>`.

7.1.1 Code Delimiters

Any code that is between the tags `<!--calypso></calypso-->` or `<calypso></calypso>` will be interpreted. Note that there can be multiple sets of such tags within a document.

All text outside of these tags is ignored by the document parser and is treated as regular HTML. All text within these tags, however, must be syntactically correct and cannot include HTML tags. The text is parsed for special directives and HTML tags will raise exception(s) unless they are included in an inline directive.

7.1.2 Logical Expressions

The following keywords are available in the language: if, else, include, set, inline, and iterator. They are described below.

Note that they are case sensitive.

The following syntax is used in this section:

- Whenever you see a definition that uses `<word>`, it means that the word expression is defined elsewhere.
- A `<statement>` can be any of the following: `<if statement>`, `<set statement>`, `<include statement>`, or `<inline statement>`. Hereafter if you see the text `<statement>`, you can substitute any of these expressions instead.
- `<statements>` is a succession of `<statement>`, typically separated with a semicolon, very much like in modern programming languages.

<if statement>

```
if ( <conditions> ) <statement>
```

or

```
if ( <conditions> ) { <statements> }
```

The start and end brackets are optional, but they are necessary if you wish to have multiple statements.

<conditions> enables you to chain various <condition> statements together using logical operators && (AND), || (OR), and ! (NOT). Hence, the following would be a valid set of conditions:

```
<condition> && ( <condition> || <condition> || ! <condition>)
```

It is also possible to parse a |KEYWORD| inside an "if" statement. For example, `if (|MASTERAG_NAME| == "ISDA" && |MASTERAG_SIGN| != "SIGNED")`, where |MASTERAG_NAME| and |MASTERAG_SIGN| are defined as keywords available from MessageFormatter.

Nested "if" statements are supported, and the "else" keyword can be added to provide a CATCH ALL clause at the end.

<condition>

A condition basically checks the value of an object attribute or perhaps the result of a method and compares it against a fixed literal value. Certain values can be returned directly from predefined objects. For more customized operations, an interface can be implemented so that a call can be made to a custom class.

The following objects are predefined: Message, Transfer, Trade, Product, Sender, and Receiver. So, for example, all the following would constitute valid condition statements:

```
Trade.quantity > 100000

Message.status = "CANCELED"

Transfer.isPayment() = true

Sender.lastName = "Johnson"

Product.getRateIndex() like "%LIBOR%"
```

As you can see, it is possible to query the field directly (quantity, status) or to actually make a method call on the related object (*isPayment()*, *getRateIndex()*). The available methods can be found by looking at the API reference for the related objects (`com.calypso.tk.core.Trade`, `com.calypso.tk.bo.BOTransfer`, `com.calypso.tk.bo.BOMessage`, `com.calypso.tk.core.Product`, and `com.calypso.tk.refdata.LEContact`).

You can also call custom functions as described in How to Create Custom Functions, with the following syntax:

```
MyFunction("arg1", "arg2") > 0.75
```

As far as available comparisons, here are all the valid operators: **<, >, <=, >=, == (equals), != (not equal), like, in, and notin**. The like operator works identically to that found in SQL.

<set statement>

```
set KEYWORD = "value";
```

All the values for identifiers used in set statements can be used as default values. If a keyword is undefined or its value cannot be extracted otherwise by MessageFormatter, the 'default' value could be retrieved from the 'set' directive.

Take this code snippet for example:

```
<calypso>
set HELLO="Bonjour";
</calypso>
...
<center>|HELLO|</center>
...
```

In this case, the HTML output for keyword HELLO will default to Bonjour unless it has been overridden elsewhere (perhaps there is a *parseHELLO()* method that does the job.) In any case, this provides a convenient method to set default values for keywords right there in the document. Note that set statements can also be used in conditions:

```
<calypso>
if ( Message.language == "English" )
    set HELLO="Hello";
if ( Message.language == "French" )
    set HELLO="Bonjour";
```

```
</calypso>
```

Also, you can use the set statement to store function results, as shown in the example below:

```
<!--calypso>
set TRADE ID = Trade.getId();
set PRODUCT TYPE = Trade.getProductType();
set CUSTOM VALUE = MyCustomFunction("One", "Two", "Three");
</calypso-->
...
We are sending you this |PRODUCT TYPE| Trade Confirmation for Trade ID |TRADE ID|.
Here is the custom value:  |CUSTOM VALUE|.
...
```

Note that in order for CUSTOM_VALUE to be set, the parser expects the class `tk.bo.formatter.MyCustomFunction` to exist. See How to Create Custom Functions for details.

<include statement>

An include statement reads and inserts a text specified in the URL string into the generated document.

```
include "<url>";
```

<url> can be a filename "myfile.html" located in the template directory, or any valid URL (for example, http://www.mysite.com/myfile.html).

<inline statement>

An inline statement inserts the text within quotation marks directly into the generated document.

```
inline "HTML text";
```

For example:

```
if ( Trade.quantity == 0 )
    inline "<b>Trade quantity is 0.</b>";
```

Note - You cannot escape the " character in an inline statement. In other words, the following comment will cause a parsing error:

```
Inline "This is a \"String\"";
```

You probably shouldn't use the inline statement unless it is only a few lines of text in any case.

<iterator statement>

You can define iterators as in the example shown below.

```
<!--calypso>
iterator ( "CashFlow" )
   inline "
<tr>
   <td>|CASHFLOW START DATE|</td>
   <td>|CASHFLOW END DATE|</td>
   <td>|CASHFLOW RATE|</td>
</tr>
   ";
</calypso-->
```

The following iterators are provided out-of-the-box:

- BondCashFlow
- BondCallSchedule
- CashFlow
- CompoundPeriod
- Fee
- MessageGroup
- PayFee and ReceiveFee

- PayLegCompoundPeriod and ReceiveLegCompoundPeriod
- PayLegFlow and ReceiveLegFlow
- StructuredProduct

7.1.3 How to Create Custom Functions

Although conditions can retrieve properties for the most commonly used objects (Product, Trade, Transfer, etc.) it is sometimes necessary to have a more custom function derive the value. This is where custom functions come in.

Custom classes can be called directly from the FormatterParser if they are placed in the `calypsox.tk.bo.formatter` or `com.calypso.tk.bo.formatter` packages and by implementing the FormatterFunction interface. This interface defines one method:

```
public Object call(DSConnection dsCon,
      BOMessage message,
      BOTransfer transfer,
      Trade trade,
      LEContact sender,
      LEContact receiver,
      Vector args);
```

For example, a custom class called MyFunction is placed in the `calypsox.tk.bo.formatter` package and this new class implements FormatterFunction.

The following code is inserted in the template file:

```
<calypso>
if ( MyFunction("one", 2) == true )
    include "subdocument";
</calypso>
```

FormatterParser will locate and find the class MyFunction and make a call to its call method as defined above. The args parameter will be a vector with 2 elements representing the arguments "one" and 2.

In this case, the semantics suggest that the *call()* method should return a Boolean object since we're comparing against a true value. Of course, there's no real way to check usage so this cannot be enforced.

Note - If an error is encountered during interpretation of conditions (for example, that a function call returns a String and when you are trying to compare with a Boolean), the condition will default to false. Hence, it is always a good idea to branch on true (in other words, not using the ! (NOT) operator is a good idea.) This will ensure that if any error is encountered at any point during the parsing, the condition will default to false and the branch will not be taken (i.e. statement(s) within the "if" branch will not be executed).

7.1.4 How to Create a Custom Display in Document Manager

In order to provide Document Security, a Document must setup regions. If a document declares no regions, then it defaults to read-only, meaning that it cannot be modified. Regions are defined using the following tags in the HTML template:

```
<!--region:NAME--> ... <!--/region-->
```

This would define a region named NAME.

Once regions are defined, permissions can be set on the documents based on the region, using Main Entry > Configuration > Messages & Matching > Document Manager. Refer to the *Calypso Messages User Guide* for details.

Calypso provides an API in order to customize how HTML documents are displayed in Document Manager.

Create a class named `tk.bo.document.CustomDocumentFilter` that implements `com.calypso.tk.bo.document.DocumentFilter`.

Sample Code

- Sample in `calypsox/tk/bo/document/CustomDocumentFilter.java`. It displays non editable regions in gray.

7.2 How to Create SWIFT Messages

A standard set of SWIFT messages are provided by the Calypso system. SWIFT messages can be generated by product type and message type, and are selected using template names, or can be generated from XML templates.

7.3 How to Create Custom Import of Message Documents

Create a class named `tk.bo.document.CustomDocumentImporter` that implements `com.calypso.tk.bo.document.DocumentImporter`.

This class will be invoked from `com.calypso.apps.reporting.MessageDocumentWindow`.

The default implementation is in `com.calypso.tk.bo.document.DefaultDocumentImporter`.

Section 8. Market Data

8.1 Quotes

8.1.1 How to use Quotes

QuoteSet is a repository for quote values used for pricing and curve generation. Typically you would obtain a QuoteSet object from a given PricingEnv.

The following example illustrates how to use QuoteSet. Specifically it will show how to obtain quote values for a given product and for curve underlying instruments used by a given curve. Furthermore it will illustrate how to manipulate quote values (bumping the quotes by 1bp) and use the bumped quotes for curve generation.

Note - There are other methods in the QuoteSet class that are not used in the example. For example there are methods specifically for getting FX quotes, rate index quotes, etc.

Sample Code

» Sample in `samples/cookbook/UseQuoteSet.java`

8.1.2 How to Subscribe to real-time Quotes

The following example illustrates how to subscribe to real-time quotes. The sample program will regenerate a given zero curve every few seconds using the latest real-time quotes.

Create a class that implements the `com.calypso.tk.marketdata.FeedListener` interface. The key method is the *newQuote()* method which is invoked whenever a real-time quote is available.

Tip

- Before running the program, make sure you have the proper real-time feed configuration in the system. Refer to the *Calypso Market Data User Guide* for information on setting up a real-time feed.

Sample Code

» Sample in `calypsox/apps/reporting/FXPositionWindow.java`

8.1.3 How to Connect to a Custom Feed Source

Calypso provides an API to extend the system to support any real-time feed source. The following example demonstrates how to connect to a real-time feed source.

Overview of Steps

- Step 1 — Create a FeedHandler
- Step 2 — Register the new FeedHandler

Step 1 — Creating a FeedHandler

Create a class named `tk.marketdata.<feed_name>FeedHandler` which extends `com.calypso.tk.marketdata.FeedHandler`.

The FeedHandler is responsible for creating a physical connection to the real-time feed and provide mapping between Calypso quote names and feed quote names based on the Feed Address Config.

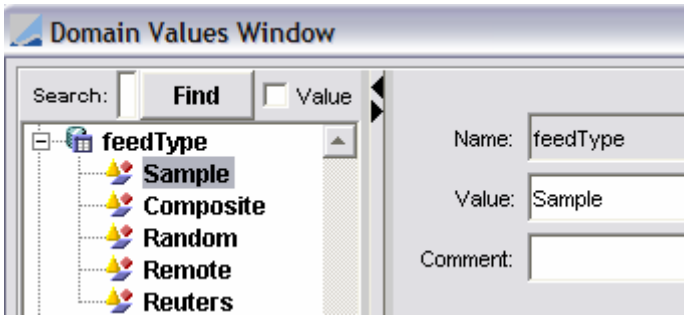
This class will be invoked from `com.calypso.tk.marketdata.FeedHandler`.

Sample Code

» Samples in `calypsox/tk/marketdata/SampleFeedHandler.java` and `com/calypso/tk/marketdata/RandomFeedHandler.java`

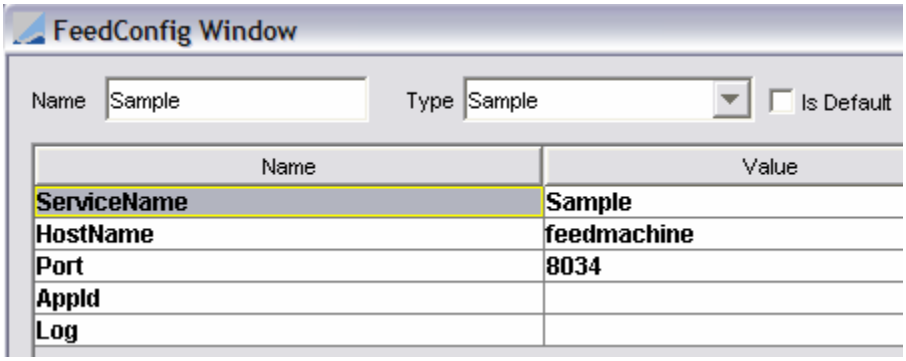
Step 2 — Registering the new FeedHandler

The feed must first be registered with the system before it can be used. To do so, add the feed name to the feedType domain.

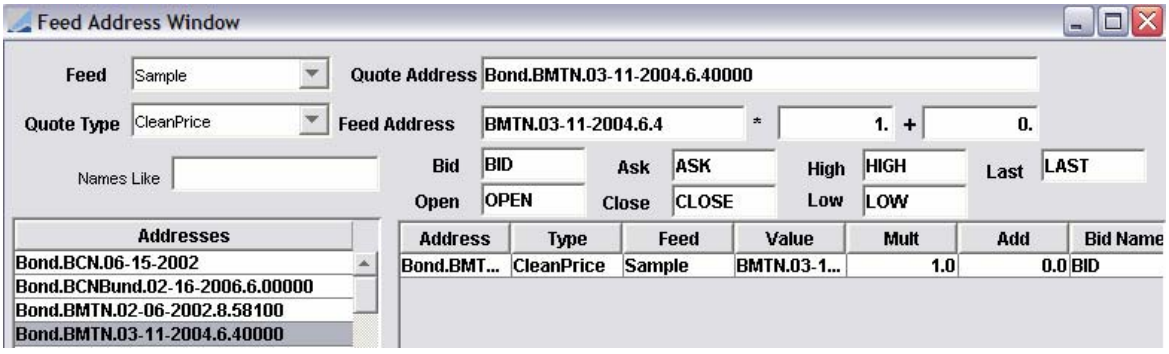


Then, you need to configure the feed:

- Specify connection parameters using Main Entry > Configuration > Market Data > Feed Configuration as shown in the example below. Check the "Is Default" box if you want to use this feed as the default feed.



- Map Calypso quote names to the feed's quote names using Main Entry > Configuration > Market Data > Feed Address Mapping as shown in the example below.



Refer to the *Calypso Market Data User Guide* for information on setting up a real-time feed.

You can inspect real-time quotes using Main Entry > Market Data > Market Quotes > Feed Quotes.

8.2 Market Data Items

8.2.1 How to Create a Custom Curve

Do the following for creating a custom curve:

1. Create a class named `tk.marketdata.<curve_type>` which extends the abstract base class `com.calypso.tk.marketdata.Curve`.

This class will be invoked from `com.calypso.tk.marketdata.Curve`.

2. To make the curve persistent, create a class named `tk.marketdata.sql.<curve_type>SQL` which extends the abstract base class `com.calypso.tk.marketdata.sql.MarketDataItemSQL`.

This class will be invoked from `com.calypso.tk.marketdata.sql.MarketDataItemSQL`.

3. If the curve is persistent, create a database table and a corresponding stored procedure for storing the curve's instances.

4. Register the curve into the marketDataType domain.

8.2.2 How to Populate a Curve with Quotes

A sample program shows how to populate a curve with quotes: `samples/ImportCurveProbability.java`.

8.2.3 How to Create a Custom Volatility Surface

Do the following for creating a custom volatility surface:

1. Create a class named `tk.marketdata.<vol_surface_type>` which extends the class `com.calypso.tk.marketdata.VolatilitySurface3D`.

This class will be invoked from `com.calypso.tk.marketdata.VolatilitySurface3D`.

2. To make the volatility surface persistent, create a class named `tk.marketdata.sql.<vol_surface_type>SQL` which extends the abstract base class `com.calypso.tk.marketdata.sql.MarketDataItemSQL`.

This class will be invoked from `com.calypso.tk.marketdata.sql.MarketDataItemSQL`.

3. If the volatility surface is persistent, create a database table and a corresponding stored procedure for storing the volatility surface's instances.

4. Register the volatility surface into the marketDataType domain.

8.2.4 How to make a Custom Market Data Item Available for Selection

You can make your market data item available in the market data selector for loading, saving and deleting.

Create a class named `apps.marketdata.<market_data_type>Selector` which implements the interface `com.calypso.apps.marketdata.MarketDataItemSelector`.

This class will be invoked from `com.calypso.apps.marketdata.MarketDataUtil`.

8.2.5 How to Display a Custom Market Data Item

Create a class named `apps.marketdata.<market_data_type>Window` which implements the interface `com.calypso.apps.marketdata.MarketDataItemViewer`.

This class will be invoked from `com.calypso.apps.marketdata.MarketDataUtil`.

8.2.6 How to add a Custom Menu Item to a Curve Window

Create a class named `apps.marketdata.CustomCurveMenu<name>` which implements the interface `com.calypso.apps.marketdata.CustomCurveMenu`.

This class will be invoked from the Curve windows.

8.2.7 How to add a Custom Menu Item to the VolatilitySurface3D Window

Create a class named `apps.marketdata.CustomVolSurfaceMenu<generator_name>` which implements the interface `com.calypso.apps.marketdata.CustomVolSurfaceMenu`.

This class will be invoked from `com.calypso.apps.marketdata.VolatilitySurface3DWindow`.

8.3 Curve Generation

8.3.1 How to Create a Custom Curve Interpolator

Create a class named `tk.core.<name>` which extends the abstract base class `com.calypso.tk.core.Interpolator`.

This class will be invoked from `com.calypso.tk.marketdata.Curve`.

Sample Code

» Sample in `calypsox/tk/core/InterpZeroDEMO.java`

8.3.2 How to Create a Custom Curve Generation Algorithm

The Calypso framework allows new curve generation algorithms be added to the system. To allow for maximum flexibility, the framework allows additional input and output values to be added to the curve for curve generation. The system has built-in capabilities to save, retrieve and display any new input and output value required. Specifically, the system can accommodate the following extensions for input and output values:

- Input parameters
- Adjustments to quote values
- Adjustments to curve points

There are various ways in which adjustments to curve points can be used. For example, a pricer can use the adjustment values in pricing or a new curve point interpolator can use the adjustment values when interpolating curve points.

In this example we will create a generation algorithm for a zero curve which will make use of the input/output extensions. The curve generation algorithm will require alpha and beta for input parameters, correlation for quote value adjustment, and coefficient for curve point adjustment. The algorithm will create a curve point for each underlying instrument with value equals (1+alpha)*(1+beta) * quote value of underlying instrument. The coefficient for curve point is calculated by quote value of underlying instrument * correlation value for the quote.

Overview of Steps

- Step 1 — Create a CurveGenerator
- Step 2 — Register the new CurveGenerator

Step 1 — Creating a CurveGenerator

Create a class named `tk.marketdata.CurveGenerator<name>` which extends the abstract base class `com.calypso.tk.marketdata.CurveGenerator`.

Note - For zero curves, the CurveGenerator class should extend `com.calypso.tk.marketdata.CurveGeneratorZero`.

The *usesQuoteAdjustment()* method allows a curve generator to specify whether or not to display adjustment columns. The curve window does not display adjustment columns if the method returns false. The method returns true by default.

The *notifyChange()* method will regenerate the curve if the underlying has changed.

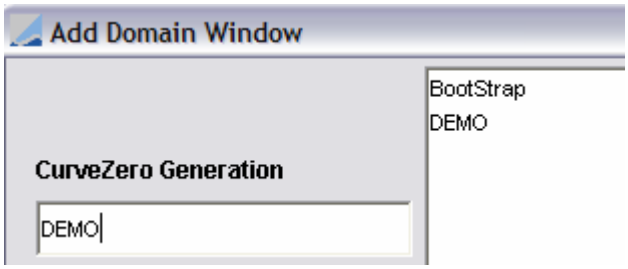
This class will be invoked from `com.calypso.tk.marketdata.Curve`.

Sample Code

- Samples in `calypsox/tk/marketdata/CurveGeneratorDEMO.java`, `calypsox/tk/marketdata/CurveGeneratorDEMO_2.java`, and `calypsox/tk/marketdata/CurveGeneratorDEMO_3.java`
- Also, `calypsox/tk/marketdata/CurveGeneratorRemote.java`, `calypsox/tk/marketdata/RemoteGenerate.java`, and `calypsox/tk/marketdata/CurveGeneratorDispRemote.java` show how to generate a curve on the server side

Step 2 — Registering the new CurveGenerator

To register the new CurveGenerator, click the ... button next to the Generation Alg field in the Curve window where you want this CurveGenerator, and add the CurveGenerator in the Add Domain window as shown below.



The new CurveGenerator will be available for selection, and the input parameters will appear under the Quotes panel of the Curve Window as applicable.

When the curve is generated, the output will contain the point adjustment coefficient under the Points panel of the Curve Window as applicable.

8.3.3 How to make Generator Parameters Persistent

Create a class named `tk.marketdata.sql.CurveGenerator<name>SQL` that implements `com.calypso.tk.marketdata.sql.CurveGeneratorSQL`.

This class will be invoked from `com.calypso.tk.marketdata.sql.CurveSQL`.

Sample Code

- Samples in `calypsox/tk/marketdata/CurveGeneratorDerivedTstData.java`, and `calypsox/tk/marketdata/sql/CurveGeneratorDerivedTstSQL.java`

8.3.4 How to Display Generator Parameters in a Popup Window

Note - This also applies to generator parameters for volatility surface.

The user can launch a GeneratorParameter window by double-clicking in the Parameters table of the Curve window or Volatility Surface window under the Quotes panel.

Create a class named apps.marketdata.GeneratorParameter<parameter_name> which implements the interface com.calypso.apps.marketdata.GeneratorParameter.

This class will be invoked from com.calypso.apps.marketdata.GeneratorParameterUtil.

8.3.5 How to Create a Custom Curve Underlying Instrument

Do the following for creating a custom curve underlying instrument:

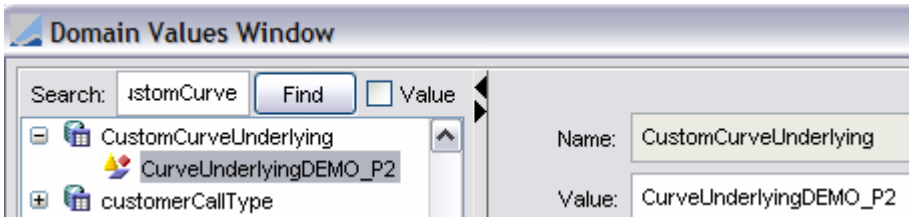
1. Create a class named tk.marketdata.CurveUnderlying<instrument_type> which extends the abstract base class com.calypso.tk.marketdata.CurveUnderlying.

This class will be invoked from com.calypso.tk.marketdata.CurveUnderlying.

2. To make the curve underlying instrument persistent, create a class named tk.marketdata.sql.CurveUnderlying<instrument_type>SQL which extends the class com.calypso.tk.marketdata.CUSQL.

This class will be invoked from com.calypso.tk.marketdata.sql.CurveUnderlyingSQL.

3. Register the new curve underlying instrument in the CustomCurveUnderlying domain.



Sample Code

- CurveUnderlying samples in calypsox/tk/marketdata/CurveUnderlyingDEMO_P1.java and calypsox/tk/marketdata/CurveUnderlyingDEMO_P2.java
- CUSQL samples in calypsox/tk/marketdata/sql/CurveUnderlyingDEMO_P1SQL.java and calypsox/tk/marketdata/CurveUnderlyingDEMO_P2SQL.java

8.3.6 How to Display a Custom Curve Underlying Instrument

Create a class named apps.marketdata.CU<instrument_type>Panel which implements the interface com.calypso.apps.marketdata.CUPanel.

This class will be invoked from com.calypso.apps.marketdata.CUWindow, which will display a panel for the new curve underlying instrument.

Sample Code

- Samples in calypsox/apps/marketdata/CUDEMO_P1Panel.java and calypsox/tk/marketdata/CUDEMO_P2Panel.java

8.3.7 How to use a Custom Curve Underlying Instrument for Curve Generation

Sample Code

We will now create the curve generation algorithm DEMO-2 that supports the curve underlying instrument DEMO_P2. CurveGeneratorDEMO-2 extends CurveGeneratorDEMO created above, and redefines the method *getCurveUnderlyingTypes()*. When the CurveGenerator DEMO-2 will be selected, DEMO_P2 will be available as underlying instrument.

» Sample in calypsox/tk/marketdata/CurveGeneratorDEMO-2.java

8.4 Volatility Surface Generation

8.4.1 How to Create a Custom Volatility Surface Interpolator

Create a class named tk.core.<interpolator_name> which extends the abstract base class com.calypso.tk.core.Interpolator3D.

This class will be invoked from com.calypso.tk.marketdata.MarketDataSurface and com.calypso.tk.marketdata.VolatilitySurface3D.

8.4.2 How to Create a Custom Volatility Surface Generation Algorithm

Overview of Steps

- Step 1 — Create a VolSurfaceGenerator
- Step 2 — Register the new VolSurfaceGenerator

Step 1 — Creating a VolSurfaceGenerator

Create a class named tk.marketdata.VolSurfaceGen<name> which extends the abstract base class com.calypso.tk.marketdata.VolSurfaceGenerator.

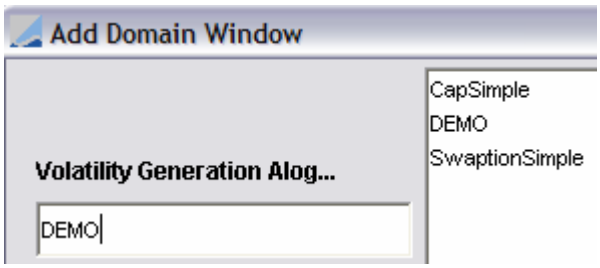
This class will be invoked from com.calypso.tk.marketdata.MarketDataSurface and com.calypso.tk.marketdata.VolatilitySurface3D.

Sample Code

» Samples in calypsox/tk/marketdata/VolSurfaceGenDEMO.java, calypsox/tk/marketdata/VolSurfaceGenDEMO_2.java and calypsox/tk/marketdata/VolSurfaceGenVega.java

Step 2 — Registering the new VolSurfaceGenerator

To register the new VolSurfaceGenerator, click the ... button next to the Generation field in the Volatility Surface window where you want this VolSurfaceGenerator, and add the VolSurfaceGenerator in the Add Domain window as shown below.



The new VolSurfaceGenerator will be available for selection.

Note that for FX volatility surface generators, derived generators are registered in the domain "FXVolSurfaceGenerator", and simple generators are registered in the domain "FXVolSurface.gensimple".

8.4.3 How to make Generator Parameters Persistent

Create a class named `tk.marketdata.sql.VolSurfaceGenerator<name>SQL` that implements `com.calypso.tk.marketdata.sql.VolSurfaceGeneratorSQL`.

This class will be invoked from `com.calypso.tk.marketdata.sql.VolatilitySurface3DSQL`.

Sample Code

- Samples in `calypsox/tk/marketdata/sql/VolSurfaceGeneratorCapCurveSQL.java`, and `calypsox/tk/marketdata/VSGenCapCurveData.java`

8.4.4 How to Display Generator Parameters in a Popup Window

See here.

8.4.5 How to Create a Custom Volatility Surface Underlying Instrument

Do the following for creating a custom volatility surface underlying instrument:

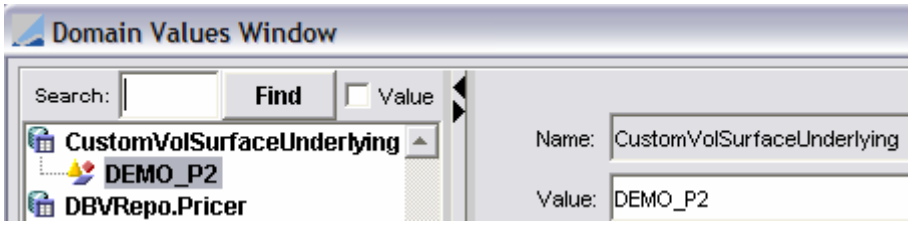
1. Create a class named `tk.marketdata.VolSurfaceUnderlying<instrument_type>` which extends the abstract base class `com.calypso.tk.marketdata.VolSurfaceUnderlying`.

This class will be invoked from `com.calypso.tk.marketdata.VolSurfaceUnderlying`.

2. To make the volatility surface underlying instrument persistent, create a class named `tk.marketdata.sql.VolSurfaceUnderlying<instrument _type>SQL` which extends the class `com.calypso.tk.marketdata.sql.VolUSQL`.

This class will be invoked from `com.calypso.tk.marketdata.sql.VolSurfaceUnderlyingSQL`.

3. Register the new volatility surface underlying instrument in the CustomVolSurfaceUnderlying domain.



Sample Code

- VolSurfaceUnderlying sample in `calypsox/tk/marketdata/VolSurfaceUnderlyingDEMO_P2.java`
- VolSurfaceUnderlyingSQL sample in `calypsox/tk/marketdata/sql/VolSurfaceUnderlyingDEMO_P2SQL.java`

8.4.6 How to Display a Custom Volatility Surface Underlying Instrument

Create a class named `apps.marketdata.VolUnderlying<instrument_type>Panel` which implements the interface `com.calypso.apps.marketdata.VolUnderlyingPanel`.

This class will be invoked from `com.calypso.apps.marketdata.VolUnderlyingWindow`, which will display a panel for the new volatility surface underlying instrument.

Sample Code

- Sample in calypsox/apps/marketdata/VolUnderlyingDEMO_P2Panel.java

8.5 How to Create a Custom Volatility Type

Create a class named tk.marketdata.VolatilityType<name> that implements com.calypso.tk.marketdata.VolatilityType.

This class will be invoked from com.calypso.tk.marketdata.VolatilityType.

8.6 How to Create a Custom Correlation Type

Create a class named tk.marketdata.CorrelationType<name> which extends the abstract base class com.calypso.tk.marketdata.CorrelationType.

This class will be invoked from com.calypso.tk.marketdata.CorrelationType.

8.7 How to Create Custom Selection Criteria for Filter Sets

Create a class named tk.marketdata.CustomFilter which implements the interface com.calypso.tk.marketdata.CustomFilterInterface.

This class will be invoked from com.calypso.tk.marketdata.FilterElement.

Sample Code

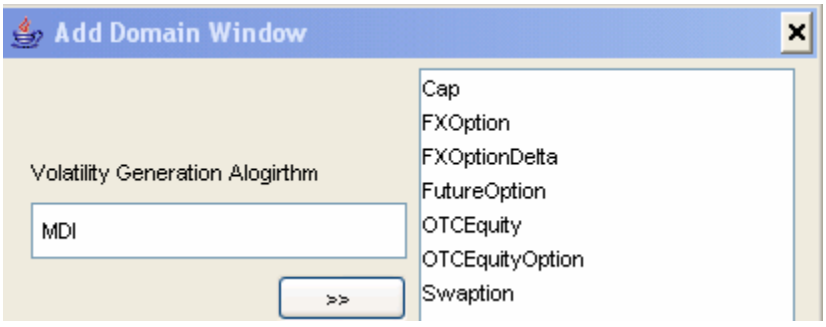
- Sample in calypsox/tk/marketdata/CustomFilter.java adds trade maturity date as a range between two tenors

8.8 How to Store Underlying Market Data with a Volatility Surface

Write a custom generator in calypsox which implements the methods *getMDIParameterNames* and *getMDIParameterType(String s)*.

Sample Code

- Sample in calypsox/tk/marketdata/VolSurfaceGenMDI.java
- Restart Main Entry and, in the Volatility Surface window, register the custom generator by clicking the ... button next to the Generation field in the Volatility Surface window where you want this Generator, and add MDI in the Add Domain window as shown below.



8.9 Pricer Configuration

8.9.1 How to Extend the Pricer Config Custom Panel

Create a class named `tk.product.<product_type>MDataSelector` that implements the interface `com.calypso.tk.product.ProductMDataSelector`.

This class will be invoked from `com.calypso.apps.marketdata.PCProductSpecificMDPanel2`.

Sample Code

- Samples in `calypsox/tk/product/BondMDataSelector.java` and `calypsox/tk/product/SwapMDataSelector.java`

Section 9. Product and Cashflow

9.1 How to Create a Custom Product

Note - When creating a custom product, you should also add a pricing model, a trade entry window, and a BOProductHandler as a minimum. Refer to the corresponding sections for details.

9.1.1 Creating a Custom OTC Product

Overview of Steps

- Step 1 — Create a Product class that captures the definition of the product
- Step 2 — Create a ProductSQL class to store the product to the database
- Step 3 — Registering the product

Step 1 — Creating a Product Class

Create a class named `tk.product.<product_type>` which extends the abstract base class `com.calypso.tk.core.Product`.

Sample Code

» Sample in `calypsox/tk/core/DEMO_P1.java`. It models a discount money market instrument containing a start date, end date, interest rate and final payment amount. The currency and the daycount of the instrument are hard-coded to USD and 30/360 respectively.

Step 2 — Creating a ProductSQL Class

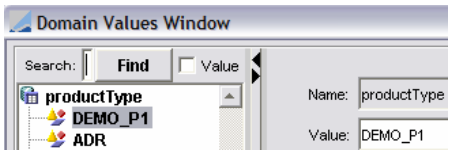
Create a class named `tk.product.sql.<product_type>SQL` which extends the abstract base class `com.calypso.tk.core.sql.ProductSQL`.

Sample Code

» Samples in `calypsox/tk/product/sql/DEMO_P1SQL.java`

Step 3 — Registering the Product

Register the new product in the productType domain.



9.1.2 Creating a Custom Exchange Traded Product

Adding an exchange traded product is similar to adding an OTC product with the exception that a product definition window has to be created. See How to Create a Custom Product Window for details.

Sample Code

» Product sample in `samples/cookbook/tk/product/DEMO_P2.java`

» Product SQL sample in `samples/cookbook/tk/product/sql/DEMO_P2SQL.java`

» Product definition window sample in `samples/cookbook/apps/product.DEMO_P2Window.java`

Note that the SQL script `calypsox/sql/demo_<dbname>.sql` contains the commands to create the necessary database tables and insert the appropriate domain values for the above product sample.

9.1.3 Creating Custom Tables for Storing Products

Create a class named `tk.product.sql.CustomSecurityTable` that implements `com.calypso.tk.core.sql.SecurityTable`.

This class will be invoked from `com.calypso.tk.core.sql.ProductSQL`.

9.1.4 Validating Security Codes for Custom Products

Call *Product.checkSecConstraints()* on the server side or *RemoteProduct.checkSecConstraints()* on the client side, to validate the security codes prior to saving. This will return a vector of error messages if any.

9.2 How to Customize Structured Products

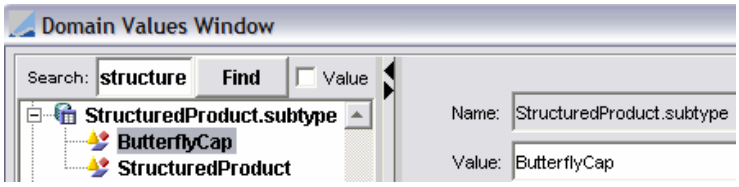
9.2.1 Creating a Custom Structured Product

Structured Product allows new product types that are composed of individual products to be added to the system in a timely manner. A Structured Product is treated as a single trade by the system. For instance, a risk report will show a structured product trade as a single trade and the back office will produce a single confirmation.

In this sample, we will add a butterfly cap, which is composed of one long cap at strike x, two short caps at strike y and one long cap at strike z with x < y < z. The notional of all the caps should be of the same amount. The trade entry validation sample code will check that the notional amount in all the component caps are equal and will issue a warning if the validation fails. The entry validation is performed before a structured product trade is saved.

Do the following for creating a custom structured product:

1. Register the new Structured Product type in the "StructuredProduct.subtype" domain.



2. Structure the product using Main Entry > Trade > Structured Product. Select ButterflyCap from the Type field. Create a butterfly cap structure by adding two long caps and two short caps.

9.2.2 How to Customize Validation by Product Subtype

You can customize the structured product validation by product subtype. Create a class named `tk.product.StructuredProduct<product_subtype>Constraint` that implements `com.calypso.tk.product.StructuredProductConstraint`.

Sample Code

» Sample in `calypsox/tk/product/StructuredProductCorridorCapConstraint.java`

9.2.3 How to Customize Report Style by Product Subtype

In the reports, you can customize the display by product subtype. The system will first look if there is a customization by product subtype, for example StructuredProductCorridorCapReportStyle, then by product type StrcturedProductReportStyle.

Sample Code

- Sample in `calypsox/tk/report/StructuredProductCorridorCapReportStyle.java`

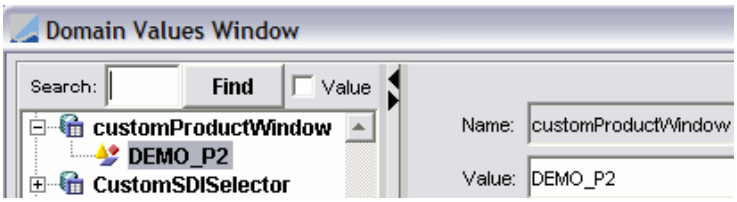
9.3 How to Create a Custom Product Window

Do the following to create a custom product window:

1. Create a class named `apps.product.<product_type>Window` or `apps.product.<product_family>Window` which implements the interface `com.calypso.apps.product.ShowProduct`.

This class will be invoked from `com.calypso.apps.main.MainEntryJFrame`.

2. Register the new window in the customProductWindow domain.



Sample Code

- Sample in `calypsox/apps/product/DEMO_P2Window.java`

  Note that the SQL script `calypsox/sql/demo_<dbname>.sql` contains the commands to create the necessary database tables and insert the appropriate domain values for the above product samples.

9.4 How to Customize Existing Products

9.4.1 Creating Custom Attributes

Do the following for creating custom attributes for a product:

1. Create a class named `tk.product.<custom_data_class_name>` which contains all of your additional attributes and which implements the interface `com.calypso.tk.core.ProductCustomData`.

2. To make the custom attributes persistent, create a class named `tk.product.sql.<custom_data_class_name>SQL` which extends the abstract base class `com.calypso.tk.core.sql.ProductCustomDataSQL`.

This class will be invoked from `com.calypso.tk.core.sql.ProductSQL`.

Sample Code

- ProductCustomData sample in `calypsox/tk/product/RepoProductExtension.java`
- ProductCusomtDataSQL sample in `calypsox/tk/product/sql/RepoProductExtensionSQL.java`

9.4.2 Using Product-Related Interfaces

SpecificResetBased

To handle specific resets, a product implements the interface SpecificResetBased. This interface requires the implementation of the method *getSpecificResets()*, which returns a vector of ProductReset.

The pricers use the vector of ProductReset to calculate cashflows known interest amounts.

9.4.3 Creating a Custom Trade Decomposition Routine

For example, you want to decompose a complex trade into more than one basic trade so that risk, positions, etc can be computed on the basic trades.

Create a class named `tk.mo.<product_type>Explode` which implements the interface `com.calypso.tk.mo.TradeExplode`. Implement the *explode()* method.

This class will be invoked from `com.calypso.tk.mo.TradeExplode`.

Sample Code

- Sample in `calypsox/tk/mo/FXForwardTakeUpExplode.java`, and `calypsox/tk/mo/IROptionExplode.java`

9.4.4 Creating a Custom Retrieval Routine for a Product

For example, you may wish to import a bond from another system when a user enters its CUSIP in a trade.

Create a class named `tk.product.sql.CustomProductFinder` which extends `com.calypso.tk.core.sql.ProductFinder`.

This class will be invoked from `com.calypso.tk.core.sql.ProductSQL` when retrieving a product.

Sample Code

- Sample in `calypsox/tk/product/sql/CustomProductFinder.java`

9.4.5 Applying Custom Validation to a Product

Create a class named `tk.product.<product_type><product_subtype> ProductValidator`, `tk.product.<product_type>ProductValidator`, `tk.product.<product_family>ProductValidator`, or `tk.product.DefaultProductValidator` which extends the class `com.calypso.tk.product.ProductValidator`.

The following methods should be implemented:

- *isValidInput()* — Returns true or false depending upon whether validation succeeds or fails.
- *applyDefaults()* — Sets default values for the product if not already set, for instance holidays. Method called before a Trade is saved if validation succeeds.

This class will be invoked from `com.calypso.tk.product.ProductValidatorUtil` to validate product information and to apply default values as applicable.

Sample Code

- Samples in `calypsox/tk/product/BondAssetBackedProductValidator.java` and `calypsox/tk/product/OTCEquityOptionVanillaProductValidator.java`

9.4.6 Creating a Custom Product Description

Typically used in reports, the task station and blotters for product subtype(s) and quote names.

Create a class named `tk.product.<product_type>ProductDescriptionGenerator` or `tk.product.<product_family>ProductDescriptionGenerator` which implements the interface `com.calypso.tk.core.ProductDescriptionGenerator`.

This class will be invoked from com.calypso.tk.core.ProductDescriptionGeneratorUtil.

Sample Code

- Sample in calypsox/tk/product/BondProductDescriptionGenerator.java

9.4.7 Creating a Custom Spot Date Calculation

This is called from the Trade windows when double-clicking on the Settle Date and from the pricers where the computation of a spot date is required.

Create a class named tk.product.<product_type>SpotDateCalculator which extends the class com.calypso.tk.product.SpotDateCalculator.

This class will be invoked from com.calypso.tk.product.SpotDateCalculatorUtil.

Sample Code

- Sample in calypsox/tk/product/SwapSpotDateCalculator.java

9.4.8 Creating a Custom Basket Calculation

Create a class named tk.product.<basket_function_name> which implements the interface com.calypso.tk.product.BasketFunction.

This class will be invoked from com.calypso.tk.product.sql.SecurityBasketSQL.

9.4.9 Creating a Custom ObservedData

Do the following for creating a custom ObservedData:

1. Create a class named tk.product.<custom_data_class_name> that implements the interface com.calypso.tk.product.CustomObservedData.

2. To make the ObservedData persistent, Create a class named tk.product.sql.<custom_data_class_name>SQL that extends com.calypso.tk.product.sql.CustomObservedDataSQL.

This class will be invoked from com.calypso.tk.product.sql.ObservedDataSQL.

Sample Code

- CustomObservedData sample in calypsox/tk/product/TestCustomObservedData.java
- CustomObservedDataSQL sample in calypsox/tk/product/sql/TestCustomObservedDataSQL.java

9.4.10 Creating a Custom Payout Formula

Create a class named tk.product.util.PayOutFormula<name> that extends com.calypso.tk.product.util.PayOutFormula.

This class will be invoked from com.calypso.tk.product.util.PayOutFormula.

9.4.11 Customizing a Bond

Creating a Custom Bond

Create a class named `tk.product.<bond_type>` that extends `com.calypso.tk.product.Bond`.

Handling Bond Prices

Bond prices are handled in the BondPrice class (in the `com.calypso.tk.core` package). This class does for bond prices what the `tk.core.Amount` class does for currency amounts. That is, it allows the user to store complete information about the price, including the tick size. Such information could not be represented in a primitive data type like a double. To support the use of BondPrice, the `tk.core.Util` class now has two methods, *bondPrice2String()* and *string2BondPrice()*, to convert from and to a BondPrice object.

When creating a new BondPrice object, you must set its tick size. Use the new method *getQuoteBase()* in the Bond class to find out the tick size, and set the BondPrice's tick size accordingly. The tick size is the integer denominator in the fractional portion of the bond's price (for example, 32, 64, or 100).

Creating a Custom Dialog for the Bond Product Window

A Custom Data button is available on the Bond product window. It will invoke a class that implements CustomDataWindow.

Create a class named `apps.product.BondCustomDataWindow` that implements `com.calypso.apps.product.CustomDataWindow`.

This class will be invoked from `com.calypso.apps.product.Bond`.

9.4.12 Customizing an ETOContract

Creating a Custom ETOContract

Create a class named `tk.product.ETO<ETO_underlying_type>` that extends `com.calypso.tk.product.ETO`.

9.4.13 Customizing a FutureContract

Creating a Custom FutureContract

Create a class `named tk.product.Future<future_type>` that extends `com.calypso.tk.product.Future`.

Creating a Custom DateGenerator for a FutureContract

Create a class named `tk.product.ContractDateGenerator<custom_name>` that implements the interface `com.calypso.tk.product.ContractDateGenerator`.

This class will be invoked from `com.calypso.tk.product.FutureContract`.

Sample Code

- Sample in `calypsox/tk/product/ContractDateGeneratorTest.java`

9.4.14 Customizing a FutureOptionContract

Creating a Custom FutureOptionContract

Create a class `named tk.product.FutureOption<future_option_type>` that extends `com.calypso.tk.product.FutureOption`.

Create a Custom DateGenerator for a FutureOptionContract

Create a class named `tk.product.OptionContractDateGenerator<custom_name>` that implements the interface `com.calypso.tk.product.OptionContractDateGenerator`.

This class will be invoked from `com.calypso.tk.product.FutureOptionContract`.

Sample Code

- Sample in `calypsox/tk/product/OptionContractDateGeneratorTest.java`

9.4.15 Credit Derivatives

Some useful interfaces.

- `com.calypso.tk.pricer.PricerReferenceEntity` — Pricers which can calculate pricer measures per reference entities (issuer id and seniority). Used by credit derivatives reports.
- `com.calypso.tk.product.CreditRisky` — Any product that may have credit risk associated with it (such as CreditDefaultSwap, TotalReturnSwap, AssetSwap, Bond, etc.). Used by credit derivatives reports.
- `com.calypso.tk.product.CreditEventBased` — Any product that can be affected by credit events (such as CreditDefaultSwap). Used by the credit event application.

9.5 How to Customize the ProductChooser Window

9.5.1 Creating a Custom Panel in the ProductChooser Window

Create a class named `tk.product.<product_type>ProductChooserHandler` which extends `com.calypso.tk.product.ProductChooserHandler`.

This class will be invoked from `com.calypso.tk.product.ProductChooserHandler`.

Sample Code

- Sample in `calypsox/tk/product/BondProductChooserHandler.java`

9.5.2 Creating a Custom ProductChooser

Create a class named `apps.product.<product_family>ProductChooser` which implements `com.calypso.apps.product.ProductChooser`.

This class will be invoked from `com.calypso.apps.product.ProductUtil` to load a list of products for display in the ProductChooser window.

Note - The Product Specific panel in the Pricer Config call *ProductUtil.getChooser()*, so if a custom ProductChooser class exists, it will be invoked from the Pricer Config.

9.6 How to Print a Product

Create a class named `tk.product.<product_type>ProductPrint` that implements `tk.product.ProductPrint`.

This class will be invoked from `com.calypso.tk.core.ProductPrintUtil`.

Sample Code

» Sample in `calypsox/tk/product/FXProductPrint.java`

9.7 How to use Cashflows

When using out-of-the-box products, cashflows will be automatically generated (provided they implement the CashFlowGeneratorBased interface).

9.7.1 Cashflows Generation

The Calypso system assumes that cashflows and all of their "known" attributes are generated from contractual data, i.e. Trade and Product classes. In other words, attributes such as flow dates and known flow amounts (for example a fixed flow, or a floating flow where the reset date is in the past) should not depend on the pricing algorithm being used. Hence, the cashflows are generated and the known amounts are calculated by the Product class (*generateFlows()* and *calculate()* methods). These flows are used by the entire Calypso system including back office components, for example to generate payments – again the assumption here being that the known payments should only depend on contractual data and not the pricing algorithm being used. If a flow attribute which is assumed to be part of the contractual data (such as the payment date of a swap flow) needs to be customized, this again is done and saved as part of trade/product using the customized checkbox on the cashflow tab.

The Pricer's responsibility is to calculate the values of the requested pricer measures. It can also project valuation attributes on the cashflows, such as the projected amount of an unknown flow (for example a floating flow where the reset date is in the future) or the survival probability used for a credit default swap flow. In addition to all of the existing attributes on the cashflows which can be set by the pricer (such as the projected amount, discount factor, etc.), the pricers can also add their own attributes by implementing the *getCashFlowColumnNames()* and *getCashFlowColumn()* methods. All of these attributes are displayed on the cashflows tab of the trade windows.

In short, the Calypso system separates the cashflow generation into two parts:

- the part that generates and sets the contractually defined attributes of the cashflows, and
- the part that sets the attributes necessary for valuation.

How to create a Custom Cashflow Calculator for a Reference Index

Create a class named `tk.product.flow.<currency><index_name>Calculator` or `tk.product.flow.<index_name>Calculator` which implements the interface `com.calypso.tk.product.flow.IndexCalculator`.

Note that `<index_name>` can be the value of the rate index attribute IndexCalculator.

This class will be invoked from `com.calypso.tk.product.flow.IndexCalculatorUtil`.

How to Create a Custom Cashflow Generator for a Product

Create a class named `tk.product.flow.CashFlow<name>` that implements `com.calypso.tk.product.flow.CashFlowSimple`.

This class will be invoked from `com.calypso.tk.product.sql.CashFlowSQL`.

How to Create a Custom Coupon Period

Create a class named `tk.product.util.<product_type>PeriodGenerator`, `tk.product.util.<product_family>PeriodGenerator`, or `tk.product.util.CustomDefaultPeriodGenerator` that implements `com.calypso.tk.product.util.CustomPeriodGenerator`.

This class will be invoked from `com.calypso.tk.product.util.PeriodGenerator` to calculate coupon period dates when generating the cashflows.

How to Create a Compound Period

Create a class named `tk.product.flow.CashFlowCompound<name>` that implements `com.calypso.tk.product.flow.CashFlowCompound`.

This class will be invoked from `com.calypso.tk.product.sql.CashFlowCompoundSQL`.

9.7.2 Cashflows Display

The functionality involving columns (accessible through the GUI) is distinct from the actual generation of cashflows by products. It is made possible via the CashFlowLayout class or possibly one of its product-specific subclasses, for example SwapCashFlowLayout. Such a class converts a CashFlowSet returned by the product's *getFlows()* method to a table for GUI display, and allows users to edit cashflows and lock cashflows to prevent changes.

All flows will initially be generated and calculated independent of the column configuration in CashFlowLayout or its subclasses. However, if the values in that column have subsequently been edited and locked, the product should contain a method that uses the appropriate CashFlowLayout class using the static method *CashFlowLayout.createCashFlowLayout(Product)* to take into account the locked flows. For example, see *SwapLeg.generateAndKeepLocksFlows()* below.

```
public CashFlowSet generateAndKeepLocksFlows(boolean paySideB, JDate asOfDate)
      throws FlowGenerationException
    {
      CashFlowSet flows = getFlows(); //just to make sure it is uncompressed
        if (flows == null || flows.size() == 0) {
            generateFlows(paySideB);
            return getFlows();
        }

         CashFlowLayout cfParser = CashFlowLayout.createCashFlowLayout("Swap");
         cfParser.processCashFlows(flows, getCouponPaymentAtEndB(),
            getPrincipalActualB(), asOfDate, this);
         long idLock = getCfGenerationLocks();
         Vector colLocks = cfParser.ids2VectorNames(idLock);
         cfParser.setColumnLocks(colLocks);
         generateFlows(paySideB);
         CashFlowSet newFlows = getFlows();
         cfParser.checkBeforeApplyingLocks(newFlows);
         cfParser.applyLockedValuesToCashFlows(newFlows, getCouponPaymentAtEndB(),
            getPrincipalActualB(), this);
         setFlows(newFlows);
         return newFlows;
    }
```

On the other hand, if only the cashflow generation is different for a product, but not the actual display of the cashflows, it may be better to subclass an existing CashFlowGeneratorBase implementation, and override the flow generation methods.

How to Create a Custom Cashflow Panel

Create a class named `tk.product.util.<product_type>CashFlowLayout`, `tk.product.util.<product_family>CashFlowLayout`, or `tk.product.util.CustomCashFlowLayout` which extends the class `com.calypso.tk.product.util.CashFlowLayout`.

This class will be invoked from `com.calypso.tk.product.util.CashFlowLayout`.

You can add custom columns to a custom CashFlowLayout in the following manner:

```
final static public int XYZ = 999;         // column id
final static public String S_XYZ = 'xyz'; // column name
```

For editable columns you can use the following ids: 51, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, and 64. Editable columns ids are stored with a product to indicate which columns are locked and modified.

Thus, you have the opportunity to edit the values in the columns, provided they are editable. The custom CashFlowLayout should implement *isColumnEditable(String colName)* for editable columns.

For non editable columns it is recommended to use ids 500 and above to avoid any conflict with the base class.

Note that while a subclass and its parent cannot have overlapping column ids, overlapping ids are allowed across subclasses.

The *parseCustomFlow()* method in a custom CashFlowLayout should only parse those custom flow fields that are not known by standard cashflow types, since the *parseFlow()* method in CashFlowLayout handles parsing of the standard cashflow types.

9.7.3 Principal Schedule

How to Create a Custom Principal Schedule Generator

This could be used in conjunction with ProductCustomData to store the attributes for the generator.

Create a class named `tk.product.util.PrincipalGenerator<structure_name>` which implements the interface `com.calypso.tk.product.util.CustomPrincipalGenerator`.

This class will be invoked from `com.calypso.tk.product.util.PrincipalScheduleGenerator`.

Sample Code

- Sample in `calypsox/tk/product/util/PrincipalGeneratorSample.java`

How to Create a Custom Principal Schedule Window

Currently available in the swap, cap/floor, swaption and cash trade windows.

Create a class named `apps.product.<structure_name>PrincipalStructureDialog` which implements the interface `com.calypso.apps.product.PrincipalStructureDialog`.

How to Set and Get Amortization Parameters

To set and get amortization parameters, the following needs to be done.

1. Get the parameters Hashtable from the underlying product (we currently support SwapLeg and CapFloor): cast the product appropriately and call *getParams()*.

The returned Hashtable will be used to store and retrieve the amortization parameters.

2. To set the Start Date for example, do the following:

```
PrincipalScheduleGenerator.setStartDate(params, date);
```

where params is the previously retrieved Hashtable, date is a JDate object representing the start date to be stored.

3. To get the Start Date for example, do the following:

```
JDate date = PrincipalScheduleGenerator.getStartDate(params);
```

where params is the previously retrieved Hashtable.

The amortization schedule, amount, rate, frequency and daycount can be obtained from the SwapLeg or CapFloor directly by calling get and set methods like getAmortAmount()/setAmortAmount(amount).

Refer to the Javadoc for PrincipalScheduleGenerator, SwapLeg, and CapFloor for details.

Section 10. Pricing

10.1 Pricing Environment

10.1.1 How to use a Pricing Environment

A Pricing Environment tells the system what pricers (pricing models) and market data (interest rate curves, volatility surfaces, quotes etc.) to use to value each product. A Pricing Environment contains a PricerConfig and a QuoteSet object. The PricerConfig specifies what Pricer to use for a product and which market data items to use with the Pricer. The QuoteSet is a repository of quote values that are needed for valuation and market data generation.

Sample Code

- Sample in `samples/PricingEnvSample.java`. It shows how to price a trade given a PricingEnv.

10.1.2 How to Create a Custom Panel for the PricerConfig Window

Create a class named `apps.marketdata.PCProductSpecificMDPanel` that implements the interface `com.calypso.apps.marketdata.PCProductSpecificMDPanel`.

This class will be invoked from `com.calypso.apps.marketdata.PricerConfigWindow`.

Sample Code

- Sample in `calypsox/apps/marketdata/PCProductSpecificMDPanel.java`

10.2 Pricer

10.2.1 How to Create a Custom Pricer

Overview of Steps

- Step 1 — Create a Pricer
- Step 2 — Register the new Pricer

Step 1 — Creating a Pricer

Create a class named `tk.pricer.<pricer_name>` which extends the abstract base class `com.calypso.tk.core.Pricer`.

The Pricer calculates a number of pricer measures: NPV, DELTA, etc. Note that the Calypso framework does not limit the type or number of pricer measures that a pricing model can support.

Important Note - If the custom Pricer uses market data specified in the Product Specific, Custom or Credit panels of the Pricer Configuration, it needs to be Lazy Refresh compatible (see below for details).

This class will be invoked from `com.calypso.tk.core.Pricer`.

Sample Code

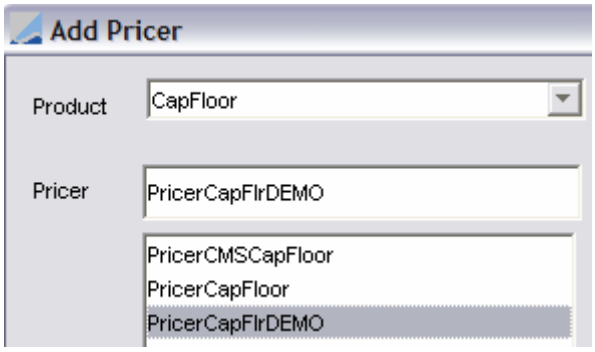
» Samples in calypsox/tk/pricer/PricerCapFlrDEMO.java and calypsox/tk/pricer/PricerDEMO_P2.java

PricerCapFlrDEMO is a pricing model for Cap/Floor. The pricing model supports the calculation of the following pricer measures:

- NPV will return zero rate * volatility at maturity date of the cap
- DELTA, GAMMA, THETA, VEGA and CASH will return 1, 2, 3, 4, and 0 respectively

Step 2 — Registering the new Pricer

Register the new pricer using Main Entry > Configuration > System > Add Pricer.



10.2.2 How to Make a Pricer Lazy Refresh Compatible

Lazy Refresh is a mode on the Pricer Configuration which improves the system performance by not loading the market data items, only their ids. The actual market data items are loaded when requested by an application, such as a Pricer. This is achieved by giving the list of MarketDataItem ids which are not yet loaded to the Pricer Configuration, and refreshing the Pricer Configuration.

The Lazy Refresh mode only applies to market data specified in the Product Specific, Custom and Credit panels of the Pricer Configuration. If a custom Pricer uses market data from these panels it MUST be made Lazy Refresh compatible, otherwise the market data will not be loaded in Lazy Refresh mode.

In order to make a Pricer compatible with the Lazy Refresh mode, you must override the following methods:

- *getMarketDataItemIds(Trade trade, PricingEnv env, JDate valDate, Hashtable itemIds)*

  In the itemIds Hashtable provide the list of lazy refresh enabled MarketDataItem ids required by the Pricer that are not yet loaded as: Key=Integer (MarketDataItem id), Value=Integer (MarketDataItem id).

- *getMarketDataItemsWithoutRefresh(Trade trade, PricingEnv env, JDate valDate, Hashtable items)*

  In the items Hashtable provide the list of MarketDataItems that are already loaded from the Pricer Configuration and can be used without refreshing the Pricer Configuration as: Key=MarketDataItem, Value=MarketDataItem.

Sample Code for Retrieving a Probability Curve

Probability curves are defined in the Credit panel of the Pricer Configuration.

```
public void getMarketDataItemIds(Trade trade, PricingEnv env,
   JDate valDate, Hashtable itemIds) {
      CreditDefaultSwap cds = (CreditDefaultSwap)trade.getProduct();
      ReferenceEntitySingle refEntity =
         (ReferenceEntitySingle)cds.getReferenceEntity();
      if (refEntity == null) return;
      PricerConfig config = env.getPricerConfig();
      Integer probCurveId = getCreditMarketDataItemId(config, cds, refEntity,
         PricerConfig.PROBABILITY);

      if (probCurveId != null) {
```

```
         itemIds.put(probCurveId,probCurveId);
      }
}

private Integer getCreditMarketDataItemId(PricerConfig config, CreditDefaultSwap cds,
   ReferenceEntitySingle refEntity, String usage) {
      Integer itemId = null;
      int tickerId = refEntity.getTickerId();
      if (tickerId > 0) {
         itemId = config.getCreditMarketDataItemId(usage, tickerId);
      }
      if (itemId != null) return itemId;
      int issuerId = refEntity.getLegalEntityId();
      String seniority = refEntity.getSeniority();
      itemId = config.getCreditMarketDataItemId(usage, cds.getCurrency(), issuerId,
         seniority, cds.getRestructuringType());
        return itemId;
}
```

Sample Code for Retrieving a Dividend Curve

Dividend curves are defined in the Product Specific panel of the Pricer Configuration.

```
public void getMarketDataItemIds(Trade trade, PricingEnv env,
   JDate valDate, Hashtable itemIds) {
      Product equityOption = trade.getProduct();
      Product underlying = getOptionUnderlying(equityOption);
      PricerConfig config = env.getPricerConfig();
      Integer divId =getDividendCurveId(equityOption, underlying, config);
      if (divId != null) {
         itemIds.put(divId,divId);
      }
}

protected Integer getDividendCurveId(Product option, Product underlying,
   PricerConfig config) {
      if (underlying == null) return null;
      return (Integer) config.getProductSpecificMDId(PricerConfig.DIVIDEND,
         underlying);
}
```

Note - In Lazy Refresh mode, *getProductSpecificMD()* will NOT return the MarketDataItem, you must use *getProductSpecificMDID()* instead, as shown above.

Custom Programs

If you have a custom program that needs to retrieve market data in Lazy Refresh mode, you need to refresh the pricer configuration to load the MarketDataItems for the MarketDataItem ids.

```
mdataItem = config.getProductSpecificMD(usage, pe);
if (mdataItem == null) {
   Integer mdataItemId = config.getProductSpecificMDID(usage, pe);
   if (mdataItemId == null) {
      // the market data is really not there
      // throw an exception that market data is missing
   }
   else {
      Vector ids = new Vector();
      ids.addElement(mdataItemId);
      // refresh the pricer configuration
      config.refresh(ids, env);
      mdataItem = config.getMarketDataItem(mdataItemId);
   }
}
```

If you want to load the full pricing environment, use *PricerConfig.getAllMarketDataItems()*. It will load all MarketDataItems, including market data specified in the Product Specific, Custom and Credit panels of the Pricer Configuration.

10.2.3 How to Create a Custom Pricing Parameter Entry Panel

Create a class which implements the interface `com.calypso.apps.trading.PricerInputViewer`. The name of the class should be returned by your pricer's *getInputViewerClassName()* method. By default `apps.trading.PricingJPanel` will be used.

This class will be invoked from `com.calypso.apps.trading.TradeViewerJFrame`.

10.2.4 How to Perform a Custom Action after Pricing

For example, you want to change the color or font of your pricing results after pricing.

Create a class named `apps.trading.PricerOutputViewer<pricer_name>` which implements the interface `com.calypso.apps.trading.PricerOutputViewer`.

This class will be invoked from `com.calypso.apps.trading.TradeWindow`.

10.2.5 How to Create a Custom Solver

Create a class named `tk.pricer.<product_family>SolveFor` or `tk.pricer.<product_type>SolveFor` that implements `com.calypso.tk.pricer.SolveFor`.

This class will be invoked from `com.calypso.tk.pricer.SolveForUtil`.

10.2.6 How to Create a Custom Inflation Forecasting Method

Create a class named `tk.pricer.<currency><index_name>InflationCalculator` or `tk.pricer.<index_name>InflationCalculator` that implements `com.calypso.tk.pricer.InflationCalculator`.

It will be invoked from `com.calypso.tk.pricer.InflationUtil`.

10.3 PricerMeasure

10.3.1 How to Create a Custom Pricer Measure

A new pricer can calculate pricer measures that are not supported by the PricerMeasure class. In this case you have to create a new PricerMeasure class.

Overview of Steps

- Step 1 — Create a PricerMeasure
- Step 2 — Register the new PricerMeasure types

Step 1 — Creating a PricerMeasure

Create a class named `<pricer_measure_class_name>` which extends the class `com.calypso.tk.core.PricerMeasure`. It is recommended to place the PricerMeasure class in the `tk.pricer` package but it is not mandatory.

Overwrite the following methods:

- *getName()*

- *toString()*
- *toInt()*
- *getDisplayClass()*
- *isAdditive()*
- *calculate()* — This method should be implemented if a given pricer measure type is applicable for all pricers across all product types. All existing pricers will invoke this method when a new pricer measure type is encountered.
- *isImplementedByPricer()*

```
public boolean isImplementedByPricer(Pricer pricer) {
   return true;
}
```

This class will be invoked from `com.calypso.tk.util.PricerMeasureUtility`.

Specify a name and an id for each PricerMeasure type as shown below.

```
final static public int ZV_SPREAD=138;
final static public int ZV_YIELD=139;
final static public String S_ZV_SPREAD="ZV_SPREAD";
final static public String S_ZV_YIELD="ZV_YIELD";
```

Tips

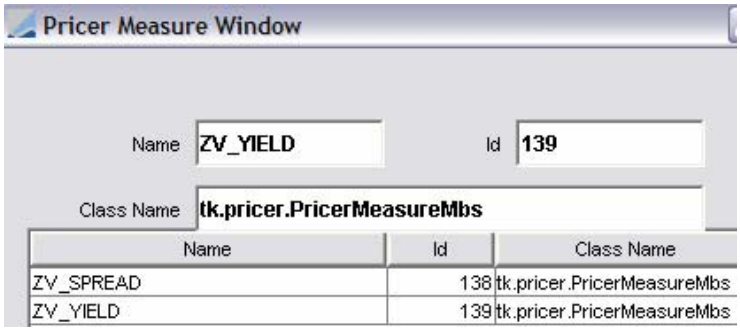
- Use an id that starts with 1000 or higher to avoid any conflict with Calypso pricer measure types.
- Create a single PricerMeasure class to hold all pricer measure types that are required by your pricing models in order to centralize the information.

Sample Code

» Samples in `calypsox/tk/pricer/PricerMeasureTst.java` and `calypsox/tk/pricer/PricerMeasureMbs.java`

Step 2 — Registering the new PricerMeasure Types

Register the PricerMeasure types using Main Entry > Configuration > System > Add Pricer Measure.



For each PricerMeasure type, enter the name, its associated id and the fully qualified class name of the PricerMeasure class.

10.3.2 How to Create Client Data for a PricerMeasure

Create a class that implements the interface `com.calypso.tk.core.PricerMeasureClientData` to specify a custom viewer for example. Your pricer creates an object of this type, and attaches it to the PricerMeasure by calling *PricerMeasure.setClientData()*.

10.3.3 How to Create a Custom Display for a PricerMeasure

Create a class named `apps.trading.PricerMeasure<viewer_name>Viewer` or `apps.trading.PricerMeasure<pricer_measure_name>Viewer` that extends Jdialog and implements `com.calypso.apps.trading.PricerMeasureViewer`. The viewer name can be set through client data.

This class will be invoked from `com.calypso.apps.trading.PricerMeasureUtil` and will display a popup window when you double-click on the PricerMeasure results. If client data has been set on this pricer measure, it will retrieve the viewer name from the client data, otherwise, it will invoke `PricerMeasure<pricer_measure_name>Viewer`.

Sample Code

- Sample in `calypsox/apps/trading/PricerMeasureNPVViewer.java`

Section 11. Trade

11.1 Trade

11.1.1 How to Create Custom Attributes for a Trade

Do the following for creating custom attributes for a Trade:

1. Create a class named `tk.core.<custom_data_class_name>` which contains all of your additional attributes and which implements the interface `com.calypso.tk.core.TradeCustomData`.

2. To make the custom data persistent, create a class named `tk.core.sql.<custom_data_class_name>SQL` which extends the abstract base class `com.calypso.tk.core.sql.TradeCustomDataSQL`.

This class will be invoked from `com.calypso.tk.core.sql.TradeSQL`.

Sample Code

- TradeCustomData sample in `calypsox/tk/core/RepoTradeExtension.java`
- TradeCustomDataSQL sample in `calypsox/tk/core/sql/RepoTradeExtensionSQL.java`

11.1.2 How to Apply Custom Validation to a Trade

Create a class named `apps.trading.<ProductTypeCode>TradeValidator` or `apps.trading.CustomTradeValidator` which implements the interface `com.calypso.apps.trading.TradeValidator` and extends `Frame` or `javax.swing.Jframe`.

The custom TradeValidator will apply to all product types for which a product-specific TradeValidator is not found. Define the following methods in your TradeValidator:

- *inputInfo()* — Displays a Dialog that lets the user input extra data.
- *isValidInput()* — Checks a trade to make sure it is ready to be saved.

This class will be invoked from `com.calypso.apps.trading.TradeViewerJFrame` to create a popup window for users to input extra data as applicable, and to validate the trade prior to saving.

Sample Code

- Samples in `calypsox/apps/trading/FXTradeValidator.java`, `calypsox/apps/trading/StraddleTradeValidator.java`, and `calypsox/apps/trading/StructuredProductButterflyCapTradeValidator.java`

11.1.3 How to Create a Custom Copy and Paste Function

You can create copy and paste functions between two trades.

Create a class `apps.trading.CopyTrade<from_product_type>2<to_product_type>` which implements the interface `com.calypso.apps.trading.CopyTrade`.

This class will be invoked from `com.calypso.apps.trading.TransferableTrade`.

Sample Code

- Sample in `calypsox/apps/trading/CopyTradeSwap2FRA.java`

11.1.4 How to Create a Custom Save As New Function

Create a class named `apps.trading.CustomSaveAsNewTrade` which implements the interface `com.calypso.apps.trading.SaveAsNewTrade`.

This class will be invoked from `com.calypso.apps.trading.TradeUtil`.

Sample Code

- Sample in `calypsox/apps/trading/CustomSaveAsNewTrade.java`

11.1.5 How to Create a Custom Keyword Validator

Create a class named `apps.trading.CustomKeywordValidator` which implements the interface `com.calypso.apps.trading.KeywordValidator`.

This class will be invoked from `com.calypso.apps.trading.TradeViewerJFrame`.

Sample Code

- Sample in `calypsox/apps/trading/CustomKeywordValidator.java`

11.1.6 How to Create a Custom Mirror Trade

Create a class named `tk.product.<product_type>MirrorHandler` or `tk.product.DefaultMirrorHandler`, which implements the interface `com.calypso.tk.product.MirrorHandler`.

This class will be invoked from `com.calypso.tk.product.MirrorHandlerUtil`.

Sample Code

- Sample in `calypsox/tk/product/SwapMirrorHandler.java`

11.2 Trade Window

11.2.1 How to Create a Custom Trade Window Title

Create a class named `apps.trading.<product_type>TradeWindowTitleGenerator` that implements `com.calypso.apps.trading.TradeWindowTitleGenerator`.

This class will be invoked from `com.calypso.apps.trading.TradeWindowTitleGeneratorUtil`.

11.2.2 How to Create Custom Default Values

Create a class named `apps.trading.Trade<product_type>DefaultValues` that implements `com.calypso.apps.trading.TradeDefaultValues`.

This class will be invoked from `com.calypso.apps.trading.TradeWindow`, and will display the corresponding default values on the trade window.

Sample Code

```
package calypsox.apps.trading;

import com.calypso.tk.core.*;
import com.calypso.tk.product.*;
import com.calypso.apps.trading.*;
```

```
public class TradeCommoditySwapDefaultValues implements TradeDefaultValues {
    public void setDefaultValues(Trade trade,ShowTrade w){
        CommoditySwap swap = (CommoditySwap)trade.getProduct();
        swap.setStartDate(JDate.getNow().addTenor(new Tenor("1Y")));
    }
}
```

11.2.3 How to Create a Custom Trade Entry Window

The Calypso Framework allows trade entry windows to be created. Once a trade window is added for a custom product, it can be used by the front office for trading and by the back office to inspect.

The Framework provides a TradeWindowBase class that implements features that are common to most trade windows: cashflows panel, trade details panel, fee panel, pricing results table, etc. With TradeWindowBase, adding a new trade entry window requires adding code that is specific to the custom product.

Do the following for adding a trade entry window:

1. Create a class named `apps.trading.Trade<product_type>Window` that extends `com.calypso.apps.trading.TradeWindowBase`.

This class will be available from the Main Entry > Trade menu.

2. In addition to the trade window class a Product panel class can be added for displaying product specific information. Create a class named `apps.trading.<product_type>TradeProductPanel` which implements the interface `com.calypso.apps.trading.TradeProductPanel`.

This class will be invoked from `com.calypso.apps.trading.TradeWindowBase`.

Sample Code

- TradeWindowBase samples in `samples/cookbook/apps/trading/TradeDEMO_P1Window.java` and `samples/cookbook/apps/trading/DEMO_P1TradeProductPanel.java`
- TradeProductPanel samples in `calypsox/apps/trading/DEMO_P1TradeProductPanel.java`, `calypsox/apps/trading/DEMO_P2TradeProductPanel.java`, `calypsox/apps/trading/DEMO_P3TradeProductPanel.java` and `calypsox/apps/trading/IROptionTradeProductPanel.java`

Note that the SQL script `calypsox/sql/demo_<dbname>.sql` contains the commands to create the necessary database tables and insert the appropriate domain values for these trade samples.

11.2.4 How to add a Custom Panel to a Trade Window

Create a class named `apps.trading.CustomTabTrade<product_type>Window` that implements `apps.trading.CustomTabTradeWindow`.

Note that _tradeDetailsPanel will call the methods *buildTrade()*, *newTrade()*, and *showTrade()* of the CustomTabTradeWindow instance.

This class will be invoked from `com.calypso.apps.trading.TradeWindow`.

Sample Code

- Samples in `calypsox/apps/trading/CustomTabTradeSwapWindow.java` and `calypsox/apps/trading/CustomTabTradeRepoWindow.java`.

11.2.5 How to Create a Custom Trade Dialog

A custom trade dialog can be implemented to enter additional details for a trade. Those details can be saved as trade keywords, therefore simplifying the trade customization.

This custom trade dialog can currently only be implemented for the Bond Front trade window and Repo Front trade window. An Info button will appear in the Trade panel that will invoke the custom dialog.

Create a class name `apps.trading.<product_type>TradeCustomDetailsDialog` or `apps.trading.TradeCustomDetailsDialog` that implements `com.calypso.apps.trading.CustomDetailsDialog`.

Sample Code

- Sample in `calypsox/apps/trading/TradeCustomDetailsDialog.java`

11.2.6 How to Create a Custom Trade Display

Create a class named `apps.trading.CustomViewTrade` which implements the interface `com.calypso.apps.trading.ViewTrade`.

This class will be invoked from `com.calypso.apps.trading.TradeWindow`.

Sample Code

- Sample in `calypsox/apps/trading/CustomViewTrade.java` to red-flag a trade if it has the keyword CustomViewTrade.

11.2.7 How to add a Custom Menu Item to a Trade Window

For a specific pricer or a specific product.

Create a class named `apps.trading.CustomTradeMenu<pricer_name>` or `apps.trading.CustomTradeMenuProduct<product_type>` which implements the interface `com.calypso.apps.trading.CustomTradeMenu`.

This class will be invoked from `com.calypso.apps.trading.TradeWindow`.

Sample Code

- Sample in `calypsox/apps/trading/CustomTradeMenuProductCancellableSwap.java`

11.2.8 How to add Custom Callbacks to a Trade Window

You can add callbacks before and/or after a trade is saved, removed, created, or priced, and before the trade window is closed.

Create a class named `apps.trading.CustomTradeWindowListener` which implements the interface `com.calypso.apps.trading.TradeWindowListener`.

This class will be invoked from `com.calypso.apps.trading.TradeWindow`.

Sample Code

- Sample in `calypsox/apps/trading/CustomTradeWindowListener.java`

11.2.9 How to Create a Custom Warning Window

Create a class named `apps.trading.CustomTradeUpdate` that implements the interface `com.calypso.apps.trading.TradeUpdate`.

This class will be invoked from `com.calypso.apps.trading.TradeWindow` when a trade is modified.

11.2.10 How to Apply Custom Validation to a Trade Template

Create a class named apps.trading.CustomTradeTemplateChecker that implements the interface com.calypso.apps.trading.TrradeTemplateChecker.

This class will be invoked from com.calypso.apps.trading.TradeWindow.

Sample Code

- Sample in calypsox/apps/trading/CustomTradeTemplateChecker.java

11.2.11 How to Create a Custom FundingTradeHandler for AssetSwap

Create a class named apps.util.CustomFundingTradeHandler which implements the interface com.calypso.apps.util.FundingTradeHandler.

This class will be invoked from com.calypso.apps.trading.AssetSwapTradeProductPanel.

Sample Code

- Sample in calypsox/apps/util/CustomFundingTradeHandler.java

11.2.12 How to Create a Custom CFD Execution Portfolio

Create a class named apps.trading.CustomCFDExecutionPortfolio that implements com.calypso.apps.trading.CFDExecutionPortfolio.

This class will be invoked from com.calypso.apps.trading.CFDTerminationWindow.

Sample Code

- Sample in calypsox/apps/trading/SampleCustomCFDExecutionPortfolio.java

11.2.13 How to Create a Custom ETO Contract Selector Window

Create a class named apps.trading.CustomContractSelector that implements com.calypso.apps.trading.ContractSelectorInterface.

This class will be invoked from com.calypso.apps.trading.ContractSelectorWindow.

Sample Code

- Sample in calypsox/apps/trading/CustomContractSelector.java

11.3 How to Apply Custom Validation to CashSettleEntryWindow

Create a class named apps.trading.CustomCashSettleEntryValidator that implements the interface com.calypso.apps.trading.CashSettleEntryValidator.

This class will be invoked from com.calypso.apps.trading.CashSettleEntryWindow.

Sample Code

- Sample in calypsox/apps/trading/CustomCashSettleEntryValidator.java

11.4 How to Apply Custom Validation to a Bundle

Create a class named apps.trading.CustomBundleValidator which implements the interface com.calypso.apps.trading.CustomBundleValidator.

This class will be invoked from com.calypso.apps.trading.TradeBundleWindow.

11.5 How to Create a Custom Blotter Trade Selector

Create a class named `apps.trading.CustomBlotterTradeSelector` which implements the interface `com.calypso.apps.trading.BlotterTradeSelector`.

This class will be invoked from `com.calypso.apps.trading.TradeBlotterPanel` to extend the list of tags available in the blotter under the "Add Trades..." button for loading a single trade. Currently, it contains Trade Id, Internal Ref, External Ref.

Sample Code

- Sample in `calypsox/apps/trading/CustomBlotterTradeSelector.java`, which adds "Counterparty".

11.6 How to Add Custom Menu Items to the Trade Blotter

You can customize the popup menu that appears when you right-click a trade.

Create a class named `apps.trading.CustomBlotterMenu` that implements `com.calypso.apps.trading.BlotterMenu`.

This class will be involed from `com.calypso.apps.trading.TradeBlotterPanel`.

Sample Code

- Sample in `calypsox/apps/trading/CustomBlotterMenu.java`

11.7 How to Apply Custom Validation to a ManualLiquidation

Create a class named `apps.trading.CustomManualLiquidationValidator` that implements the interface `com.calypso.apps.trading.CustomManualLiquidationValidator`.

This class will be invoked from `com.calypso.apps.trading.ManualLiquidationJDialog`.

Sample Code

- Sample in `calypsox/apps/trading/CustomManualLiquidationValidator.java`

  This sample also illustrates how to add access permissions on the manual liquidation process in addition to the existing business rules.

11.8 How to Create a Custom Reference Entity Selection Window

Calypso's selection dialog lets users specify reference entities based on existing issuers in the system, for credit derivatives.

Create a class named `apps.trading.CustomRefEntityChooser` which extends the class `com.calypso.apps.trading.RefEntityChooserInterface`.

11.9 How to Create a Custom BO Trade Display

Create a class named `apps.trading.CustomBOTradeDisplay` which implements the interface `com.calypso.apps.trading.BOTradeDisplay`.

This class will be invoked from `com.calypso.apps.trading.BOTradeFrame`.

Tip

- To override the PO SDI selection whenever the counterparty SDI is manually selected, implement the methods *getReceiverSDISelection()* and *getPayerSDISelection()*.
- You can also use the BOTradeDisplay interface to override the assignment of netting methods for standards SDIs as well as manual SDIs by implementing the *getNettingType()* method.

  Note - These customizations will also appear in the Netting Manager, Assign and Split windows of the Task Station.
- The method *modifyTransferRule()* allows modifying the behavior of a transfer rule when it is manually modified.

Sample Code

» Sample in `calypsox/apps/trading/CustomBOTradeDisplay.java`

11.10 How to Create a Custom Fee Calculator

Do the following to create a custom fee calculator:

1. Create a class named `tk.bo.<fee_method>FeeCalculator` which implements the interface `com.calypso.tk.bo.FeeCalculator`. Implement the following methods:

- *calculate()* — calculates the fee amount.
- *calculateInverse()* — reverts the amount computed and stored on the Fee. For example, if you have a Fee Amount of 1,000 expressed as a percentage of the notional, its purpose is to display the initial percentage amount, 2%.
- *getDescription()* – returns a fee description.

This class will be invoked from `com.calypso.tk.bo.FeeDefinition`.

2. Register the new fee calculator in the feeCalculator domain.

Sample Code

» Sample in `calypsox/tk/bo/CustomerTransferFeeCalculator.java`

11.11 Three Party Trades

11.11.1 Installing

1. Edit the file `calypsox/apps/trading/SampleThreeTradeValidator.java`.

2. Replace "SampleThreeTrade" by "CustomTrade" and save the file as `calypsox/apps/trading/CustomTradeValidator.java`.

3. Compile:

`calypsox/apps/trading/CustomTradeValidator.java`

`calypsox/apps/trading/ThreePartyJPanel.java`

`calypsox/tk/bo/workflow/rule/ThreePartyTradeRule.java`

11.11.2 Configuring

1. Add the rule ThreeParty to the available trade rules using the Workflow Config.

2. Add the necessary trade keywords (if some required keywords are not set, the system will give you an error messages when trying to save the Three Party trade).

- 3PartyType
- NumTrades
- Book, Book2, Book3 (also Bookn if you intend to use n trades)
- Location, Location2, Location3 (*same remark as book*)
- Direction, Direction2, Direction3 (*same remark as book*)
- Cpty, Cpty2, Cpty3 (*same remark as book*)
- Role, Role2, Role3 (*same remark as book*)
- SeqNo

11.11.3 Customizing the Three Party Trade

If you want to add more automatic schemes, you need to customize `ThreePartyJPanel.java` and `ThreePartyTradeRule.java`.

Section 12. Trade Lifecycle

12.1 How to Create a Custom Allocation Process

Create a class named `tk.product.<product_type>ProductAllocator` which implements the interface `com.calypso.tk.product.ProductAllocator`.

This class will be invoked from `com.calypso.tk.product.ProductAllocatorUtil`.

12.2 How to Create a Custom Corporate Actions Handler

Create a class named `tk.product.<product_type>CorporateActionHandler`, `tk.product.<product_family>CorporateActionHandler` or `tk.product.DefaultCorporateActionHandler` that implements `com.calypso.tk.product.CorporateActionHandler`.

This class will be invoked from `com.calypso.tk.product.CorporateActionHandlerUtil`.

12.2.1 How to Customize a Corporate Action Amendment

Create a class named `tk.product.CustomCATradeActionLookup` that implements `com.calypso.tk.product.CATradeActionLookup`.

The method *getTradeAction()* can change the action to be applied when a corporate action is amended. The input parameters are the original action, the trade, and the CorporateAction.

It will be invoked from `com.calypso.tk.product.CorporateActionHandlerUtil`.

Sample Code

- Sample in `calypsox/tk/product/CustomCATradeActionLookup.java`. It changes AMEND to UPDATE.

12.3 Exercise and Expiration

12.3.1 How to Create a Custom Exercise Process

Create a class named `tk.product.<product_type>Exercisable` which implements the interface `com.calypso.tk.product.Exercisable`.

This class will be invoked from `com.calypso.tk.product.OptionExerciseUtil`.

12.3.2 How to Apply Custom Validation to the Exercise Process

Create a class named `tk.product.<product_type>ExerciseValidator`, `tk.product.<product_family>ExerciseValidator`, or `tk.product.DefaultExerciseValidator` that implements `com.calypso.tk.product.ExerciseValidator`.

This class will be invoked from `com.calypso.tk.product.OptionExerciseUtil`.

Sample Code

- Sample in calypsox/tk/product/DefaultExerciseValidator.java

12.3.3 How to Apply Custom Validation to the ETOExcerciseWindow

Create a class named apps.reporting.CustomFutureOptionExerciseExpiryValidator that implements the interface com.calypso.apps.reporting.FutureOptionExerciseExpiryValidator.

This class will be invoked from com.calypso.apps.reporting.ETOExerciseWindow.

Sample Code

- Sample in calypsox/apps/reporting/CustomFutureOptionExerciseExpiryValidator.

12.3.4 How to Apply Custom Validation to the FutureExpiryWindow

Create a class named apps.reporting.CustomFutureExpiryValidator that implements the interface com.calypso.apps.reporting.FutureExpiryValidator.

This class will be invoked from com.calypso.apps.reporting.FutureExpiryWindow.

Sample Code

- Sample in calypsox/apps/reporting/CustomFutureExpiryValidator.

12.4 How to Create a Custom Price Fixing Handler

To perform any additional processing during price fixing - invoked from the Price Fixing window when the user publishes the price fixings.

Create a class apps.reporting.CustomPriceFixingHandler which implements the interface com.calypso.apps.reporting.CustomPriceFixingHandlerInterface.

This class will be invoked from the com.calypso.apps.reporting.PriceFixingFrame to perform any additional processing during price fixing.

Sample Code

- Sample in calypsox/apps/reporting/CustomPriceFixingHandler.java

12.5 How to Create a Custom Rollover Process

Currently, Calypso implements rollover for treasury products such as Loans and Deposits, Repos, and FX products. The rollover process for Calypso Fixed Income products implements FIRollOver, and for Calypso FX products ForexRollOver.

Create a class named tk.product.<product_type>RollOver which implements the interface com.calypso.tk.product.FIRollOver or com.calypso.tk.product.ForexRollOver.

This class will be invoked from com.calypso.tk.product.RollOverUtil.

12.6 Termination

12.6.1 How to Create a Custom Termination Process

Create a class named `tk.product.<product_type>Termination` or `tk.product.DefaultTermination` which implements the interface `com.calypso.tk.product.Termination`.

This class can also be used for transferring trades, provided the following methods are implemented:

- *transferTrade()* — Transfers a Trade to a new book or a new counterparty.
- *filterFlows()* — Filters flows for a Trade terminated or transferred. This function is called from the getFlows method attached to the Product when the flag addTradeFlows is set to true. The purpose of this function is to remove or add any flows based on the Termination/Transfer information. It can be redefined for each Product.
- *isTradeTransferrable()* — Returns true if the Trade can be transferred.

This class will be invoked from `com.calypso.tk.product.TerminationUtil`.

12.6.2 How to Create a Custom Termination Dialog

Create a class named `apps.product.<product_type>TerminationDialog` that implements the interface `com.calypso.apps.product.TerminationDialog`.

This class will be invoked from `com.calypso.apps.product.TerminationDialog`.

12.7 How to add Custom Menu Items to the Process Trade Window

Create a class named `apps.util.CustomProcessTradeMenu` which implements the interface `com.calypso.apps.util.CustomProcessTradeMenu`.

This class will be invoked from `com.calypso.apps.util.ProcessTradeUtil`.

12.8 How to Create a Custom Attribute Matching Mechanism

Create a class named `tk.bo.matching.<matching name>MatchingAttributes` that implements `tk.bo.matching.MatchingAttributes`.

This class will be invoked from `com.calypso.tk.bo.matching.DefaultMatchingUtil` when performing attributes matching in the Financial Matching Window.

Section 13. Reporting

13.1 Report Framework Overview

The Calypso Report Framework has been designed to clearly separate the various elements of a report:

- Report Template — The input parameters for the report.
- Report — The database query to retrieve the data.
- Report Output — Data model of the data retrieved by the report.
- Report Style — Utility to extract atomic values (columns) from the Report Output.
- Report Viewer — An interface to "render" or display the report output to the end-user.
- Report Window — A single GUI window drives all the reports. In so doing, a new report can easily be implemented and plugged into the system, immediately inheriting of all the services available in the report framework: Export to Excel, HTML, PDF, Aggregation functions, sorting, etc.

13.1.1 Defining Report Templates

The ReportTemplate and its subclasses provide the means to define search query parameters and what data the report should contain. It allows this information to be stored in the database for use in Report windows and/or in running Report Scheduled Tasks.

For example, a user wants to create a custom report with the following parameters:

- Currency
- Min and Max Amount
- Display Derivatives
- Start Date and End Date

The class would be named `tk.report.<CustomObject>ReportTemplate` and should extend ReportTemplate. It will look like:

```
public class CustomObjectReportTemplate extends ReportTemplate {
public static final String CURRENCY="Currency";
public static final String MIN_AMOUNT="MinAmount";
public static final String MAX_AMOUNT="MaxAmount";
public static final String DERIVATIVES_FLAG="DerivativesFlag";
       public void setDefaults() {
              Hashtable contents = getContents();
              contents.put(DERIVATIVES_FLAG, new Boolean(false));
}
}
```

The *setDefaults()* method allows setting default query parameters for the report.

The base class "ReportTemplate" handles the following parameters:

- Description — A free-form String description of this template
- Start Date and End Date — Start Date cutoff for the report defined as an absolute date or as a relative tenor, and End Date cutoff for the report defined as an absolute date or as a relative tenor.

  Start and end dates are used in most reports, so it makes sense to provide the parameters as defaults in the base ReportTemplate class. However, it is to the discretion of the Report as to whether these are used

or not. Most of these "input" parameters useful will only be if the Report class makes use of it; in other words, if it uses these parameter values ingenerating the query.

- Holidays — Vector of Holidays to use when calculating dates
- Business Days — Boolean flag to differentiate between Business and Calendar Days
- Columns — Columns to be displayed in the Report. The Columns parameter enables you to select which columns to display in the output report. The selection of columns depend on the report and the selection is retrieved from the ReportStyle class and for each column that is defined, the ReportStyle class will need to code the logic on how that value is to be extracted from the Report row.
- Sort Columns — Columns by which the report data should be sorted. The SortColumns parameter allows you to dictate how the data rows are to be sorted. The user can specify one or more sorting columns. For example, some reports may need to be sorted by Book, then by trade ID, and others may need to be sorted simply by date. The set of Sort Columns available is taken from the set of Columns mentioned above and it is not possible to sort on a column that is not included in the Columns parameter.
- Subheadings — Columns to be displayed as "subheadings". Some reports need to show subheadings. This only applies when sorting is being used and the columns to be shown as subheadings are chosen from those selected as sort columns. For example, to sort by Settlement Date, select the "Settlement Date" column as a subheading, this will result in the value being shown on its own row, with the rows that match its value being demarcated underneath.
- Subtotals — Columns for which subtotals should be displayed. It is possible to tag columns for which subtotals should be computed. This only applies to columns that represent numeric values such as amounts. Attempting to generate a subtotal for a Legal Entity Name, say, will result in a subtotal of 0.0. Subtotals are shown whenever a break occurs in the sorting order. If a Transfer report is being sorted by Settlement Date and subtotaled by Transfer Amount the subtotal for all transfers matching the specific settlement date will be displayed. Subtotals are reset to 0 after each break in the sorting order.
- Totals — Columns for which totals should be displayed. Totals are cumulative subtotals. The only difference between a Total and a Subtotal is that the total is not reset at each break in the sorting order but a tally is kept for each row in the report.
- Subtotal Functions and Total Functions — Aggregation functions to use in calculating subtotals and totals. For each column defined as a subtotal or total, you can use an aggregation function to calculate that subtotal. The default function is Sum but other functions are available out-of-the-box including Maximum, Minimum, and Average.
- AggregationFlag — There are times when an aggregated view of the report is required without displaying all row details. If the flag is set to true (default being false) then only an aggregated view will be displayed in the report. Only rows with subheadings, subtotals, and totals will be displayed while the specific row details will be hidden from view. Of course, the subtotals and totals will include these hidden rows; they simply will not be visible in the report.

Columns, Sort Columns, Subheadings, Totals, Subtotals, and the associated Function parameters are used to control what data is displayed, how it is sorted, and if and how it is aggregated into groups (with subheadings and subtotals.) Although how the data is displayed may differ depending upon the ReportViewer (HTML will look different from a GUI table), the actual data should remain the same across all viewers.

As far as data persistence is concerned, this is managed by the `tk.report.sql.ReportTemplateSQL` class and, since it implements the `tk.core.Attributable` interface, most of the attributes are stored in the entity_attributes table.

13.1.2 Defining Reports

The Report class is where a query is built based on the input parameters in the ReportTemplate. The query is built; a call is made to the Calypso Data server (via the remote API) and the returned data is a ReportOutput with a set of ReportRows. A ReportRow identifies the objects retrieved from the system on a per-row basis.

The reason for having the ReportRow objects (as opposed to simply passing the objects Vector itself) is because there are times when one needs to associate several different objects to one row. For example, a Message Report will sometimes have a Message, a Transfer, and a trade all attached to the same ReportRow.

Continuing with the earlier example, <CustomObject>Report would be as follows:

```
class CustomOjectReport extends Report {
       public ReportOutput load() {
              initDates();
              DefaultReportOutput output = new DefaultReportOutput(this);
              Hashtable from = new Hashtable ();
              String where = buildQuery(from);
              Vector objects = null;
              try {
                     …
                     RemoteCustom rc = ds.getRemoteCustom();

                     objects = rc.getObjects(where, from);
                     …
              }
              catch (Exception e) { … }

ReportRow[] rows = new ReportRow[objects.size()];
for (int i=0; i < rows.length; i++) {
rows[i] = new ReportRow(objects.get(i));
}
output.setRows(rows);

return output;
}

…
public String buildQuery(Hashtable from) {
              // go through the CustomObjectReportTemplate and build
              // the where query based on the values set.
              // Add the associated tables to from Hashtable
              …
       }
}
```

13.1.3 Defining Report Outputs

The core class that "handles" the report output is `com.calypso.tk.report.DefaultReportOutput`. Its purpose is not to render or display the data. This task is left to the report viewers. Its purpose is to arrange the report data which was set by the report with the *setRows()* method in a way compatible with the parameters given in the ReportTemplate (Columns, SortColumns, Subtotals, etc.)

Hence, the DefaultReportOutput class will sort the rows based on the SortColumns. It will parse through each row and calculate the subtotals and totals, and pass that information along to the ReportViewer.

13.1.4 Defining Report Styles

ReportStyles are helper classes which provide the functionality to extract column values from ReportRow objects. A ReportRow encapsulates one or more Calypso objects. It is necessary to extract a column value from the object because the column might be a direct mapping to an object field value (e.g. TRADE_ID column maps to *Trade.getId()* method). Other column values might be computed and/or generated when it is queried.

ReportStyle classes are extensible. For example, the SettlementReportStyle extends TransferReportStyle, which extends TradeReportStyle, which extends ReportStyle. There is no need to duplicate the functionality to extract a column value: by extending from a superclass you can inherit that behavior.

The ReportStyle classes provide one core method which provides all of the functionality for the class, for example:

```
  protected Object getColumnValue (ReportRow row, int columnId) {
       …
       // Extract column value (columned) from the ReportRow.
// If there is no match, you can delegate the class to have it extract the
// value.
return super.getColumnValue(row, columnId);
       }
```

The following is a more specific example of the TransferReportStyle and provides a short excerpt of how it is implemented:

```
  protected Object getColumnValue (ReportRow, row, int columnId) {
       BOTransfer transfer = (BOTransfer)row.getProperty(ReportRow.TRANSFER);

       if (columnId == ID_AVAILABLE_DATE) {
       return transfer.getAvailableDate();
       }
       else if (columnId == ID_DELIVERY_TYPE) {
       return transfer.getDeliveryType();
       }
       …
       return super.getColumnValue(row, columnId);
}
```

An additional note is required regarding TradeReportStyle. In order to place Product-specific columns in their own style classes and permit clients to extend the framework easily, TradeReportStyle proceeds to search for matching column name(s) if it does not find it in its own set of columns. First, it will iterate over product interfaces since these report styles span multiple Products. All product interfaces are defined in the productInterface domain values and the following Report Styles are packaged with Calypso: OptionReportStyle, CashSettledReportStyle, among others. If the column is not located in any of the available interface report styles, then TradeReportStyle attempts to spawn a product specific ReportStyle based on the product type (i.e., SwapReportStyle, BondReportStyle, etc.)

13.1.5 Defining Report Viewers

The Report Viewer renders the report output. Calypso currently provides the following report viewers: HTML, Excel, PDF, CSV and GUI tables. ReportWindow provides the ability to view any report implemented using the framework. A customer does not need to worry about implementing a GUI for a report, the report will be immediately available via a GUI interface.

A report viewer implements the `com.calypso.tk.report.ReportViewer` interface.

Out-of-the-box, Calypso provides the following report viewers:

- `com.calypso.tk.report.HTMLReportViewer`
- `com.calypso.tk.report.ExcelReportViewer`
- `com.calypso.tk.report.PDFReportViewer`
- `com.calypso.tk.report.CSVReportViewer`
- `com.calypso.apps.reporting.TableReportViewer`
- `com.calypso.apps.reporting.TreeTableReportViewer`
- `com.calypso.apps.reporting.PivotTableReportViewer`

13.1.6 Report Window

All reports that are built on top of the Reporting Framework share the same default GUI code. At its simplest level, a report can be added to any GUI window by adding a new ReportPanel to an existing GUI window:

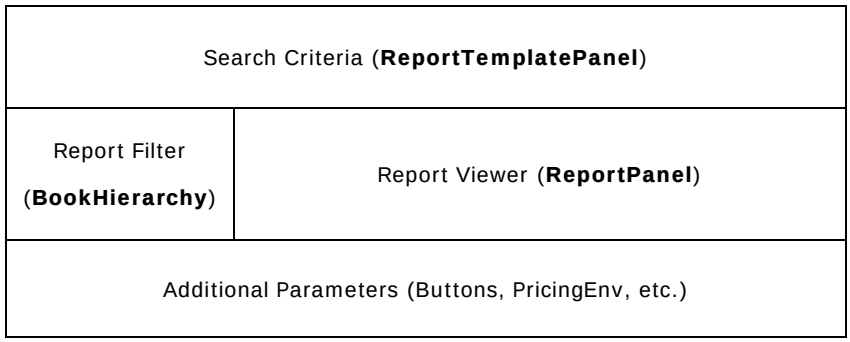
```
    // Insert a Message Report Panel
    JPanel messagePanel = new ReportPanel("Message");
```

Of course, this only provides the ability to **embed** a Report in an existing window and does not provide a way to set search criteria (ReportTemplate) on this report. This would need to be done programmatically elsewhere in the code.

For most reports, it will often be more practical to use ReportWindow:

```
     // Create a new Message Report Window and make it visible
    JFrame frame = new ReportWindow("Message");
    Frame.setVisible(true);
```

All reports embedded in the ReportWindow have a similar look and feel.



The search criteria for the reports are selected at the top in the ReportTemplatePanel. Some control buttons and additional environment settings are available at the bottom of the window in the button panel. There, you can set the Pricing Env, the Valuation Date, and connect to real-time events. These options are only available when applicable.

Given the argument passed to the Report Window ("Message" for example), the following GUI classes will be instantiated and attached to the Report Window (if and when applicable):

- `apps.reporting.MessageReportTemplatePanel`
- `apps.reporting.MessageReportWindowCustomizer`
- `apps.reporting.MessageReportRealTimeHandler`

The corresponding toolkit classes are also instantiated:

- `tk.report.MessageReportTemplate`
- `tk.report.MessageReport`
- `tk.report.MessageReportStyle`

13.1.7 Import/Export

It is possible to import/export report template parameters from/to XML. This provides a convenient way to distribute a suite of out-of-the-box reports to one or more users. The report templates can be stored in a directory and imported into the system on an as-needed basis.

Because the template parameters are stored as attributes, there should not be a need to modify or extend the import/export functionality. For reference, the relevant classes can be found in the following package: `com.calypso.bridge.object.reportTemplate`.

13.2 How to Add a New Report

In order to add a new report, create a new ReportTemplate (which may or may not extend an existing one), a new Report, and a new ReportStyle, and place them in the `tk.report` package. For example, in order to create a CustomObject report three classes would be created:

- `tk.report.CustomObjectReportTemplate`
- `tk.report.sql.CustomObjectReportTemplateSQL`
- `tk.report.CustomObjectReport`
- `tk.report.CustomObjectReportStyle`

Add the domain value ("CustomObject" in this example) to the "REPORT.Types" domain, and add a menu item for the report using Main Entry > Utilities > Main Entry Configurator as follows:

- Name — Custom Object Report
- Action — `reporting.ReportWindow$CustomObject`

After re-starting Main Entry, it will be possible to access the new report from the Reports menu, and to input the appropriate parameters, load/save templates and, run the report.

13.2.1 How to Create a Report Template Panel

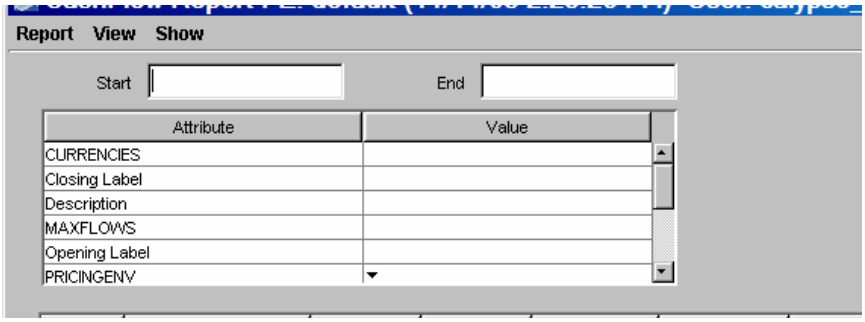
However, in order to be generic, the GUI to manipulate the ReportTemplate is not the most user-friendly. It lists most of the parameters in tabular format and does not do any kind of type-checking (Some minor type-checking is done, whenever possible, by parsing the parameter names). This is certainly not failsafe, however. The limitation here was that the ReportTemplate, which has been available in Calypso for several years, had to remain backward-compatible. Since the template is stored in the database as a Hashtable object, there is no possibility to easily upgrade it.

However, there is a way to customize how the user will interact with your template. All that is needed is to create a class named `apps.reporting.CustomObjectReportTemplatePanel` which extends `com.calypso.apps.reporting.ReportTemplatePanel`. This abstract class extends JPanel and provides two methods to set and get the ReportTemplate in order to determine how the report template should be displayed on the GUI:

- *setTemplate()*
- *getTemplate()*

This panel, once compiled, will automatically be loaded and inserted at the top of the ReportWindow.

The following ReportTemplatePanels are provided by Calypso and are used in the respective report windows. By launching these various reports, you can see the customized template GUI: AuditReportTemplatePanel, DailyBlotterReportTemplatePanel, FailsReportTemplatePanel, FeeReportTemplatePanel, SettlementReportTemplatePanel, for example. In the case of a simple, straightforward report template, the implementation of this class is not required; rather the DefaultReportTemplatePanel will be used and will display the report parameters in tabular format, as shown below:



13.2.2 How to Add a Custom Menu and Custom Processing

The report provides a convenient GUI to manipulate a vast number of objects at one time. The window can be extended by adding a menu with custom functionality to, for example, allow "manipulation" rather than simply viewing of data.

Continuing with the Custom Object report example, this would be achieved by creating a class named `apps.reporting.CustomObjectReportWindowCustomizer` that extends `com.calypso.apps.reporting.ReportWindowCustomizerAdapter`. These methods can be implemented:

- *public void customizeReportWindow(ReportWindow window)* — This is where the customization of the report can be done.
- *public boolean callBeforeClose(ReportWindowDefinition definition)* — This method returns true or false, true to authorize the report to be closed, or false to prevent the report to be closed if additional processing needs to be done like publishing quotes, etc.
- *public JMenu getCustomMenu(ReportWindow window)* — This method allows you to attach a menu to the menu bar where you can provide additional functionality. This is a convenient way to add more functionality. For example, the menu Process available in the Transfer Report is implemented in this way.

13.2.3 How to Create a Custom Aggregation Function

Create a class named `tk.report.function.<function_name>` that implements `com.calypso.tk.report.function.ReportFunction`.

It will be invoked from `com.calypso.tk.report.function.FunctionFactory`.

Then add the function name to the "REPORT.Functions" domain. Out-of-the-box, Calypso provides the following aggregation functions: Count, Sum, Average, Maximum, and Minimum.

13.2.4 How to Create a Custom Sorting Comparator

Create a class named `tk.report.<comparator_name>Comparator` that implements `com.calypso.tk.report.RowComparator`.

It will be invoked from `com.calypso.tk.report.DefaultReportOutput`.

13.2.5 How to Validate a Custom Report Filter

Create a class named `apps.reporting.<report_name>CustomReportFilter` that implements the interface `com.calypso.apps.reporting.CustomReportFilter`.

This is currently supported by the following reports: Audit, CRE, Message, Payment, Posting, Task Station and Trade.

It is also supported by the report framework through `com.calypso.apps.reporting.ReportPanel`.

Sample Code

- Multiple samples in `calypsox.apps.reporting`: AuditCustomReportFilter, CreCustomReportFilter, MessageCustomReportFilter, PaymentCustomReportFilter, PostingCustomReportFilter, TaskCustomReportFilter, and TradeCustomReportFilter.

13.3 How to Customize the Transfer Viewer

Note that the Transfer Viewer is a utility that is only available if the environment property USE_TRANSFER_VIEWER is true. When you right-click a transfer, choose Show > Transfer Viewer from the popup menu to display all elements associated with the transfer.

The Transfer Viewer can be customized as follows:

- Adding panels.
- Displaying more data in the Main panel.

Create a class named `apps.reporting.CustomTransferViewerWindow` that implements `com.calypso.apps.reporting.TransferViewerInterface`.

You can implement the following methods:

- *getTransferMainPanel()* — for customizing of the main panel.
- *getTransferTabPanels()* — for adding a panel.
- *showTransfer()* — for displaying additional transfer data.

13.4 How to Customize the Quick Search Window

You can customize the search criteria, the searched objects, and the display of the objects.

Create a class named `apps.reporting.<name>Interface` that implements `com.calypso.apps.reporting.QuickSearchInterface`.

Sample Code

» Sample in `calypsox/apps/reporting/QuickSearchWindowInterface`

13.5 How to Enable Generic Comments for an Object

To enable generic comments for an object, create a class named `tk.bo.sql.DefaultCommentableObjectSQL` that implements `tk.bo.sql.CommentableObjectSQL`.

It will be invoked from the BackOffice RMI server.

This interface must implement the two following methods:

- public ObjectDescription getCommentableObject(int objectId, String objectClass, Connection dbCon) throws PersistenceException;

  objectClass: "Trade", "Transfer", "Message", "Posting", "YourObjectClass", etc.

- public GenericComment[] getObjectComments(ObjectDescription objectDesc, Int showType, String whereClause, Connection dbCon) throws PersistenceException;

  showType: GenericComment.SHOW_ALL, GenericComment.SHOW_PARENTS, GenericComments.SHOW_CHILDREN, GenericComment.SHOW_ALL, GenericComment.SHOW_NONE (no "Show" directive)

Sample Code

» Sample in `calypsox/tk/bo/sql/DefaultCommentableObjectSQL.java`

Section 14. Risk Analysis

14.1 Analysis

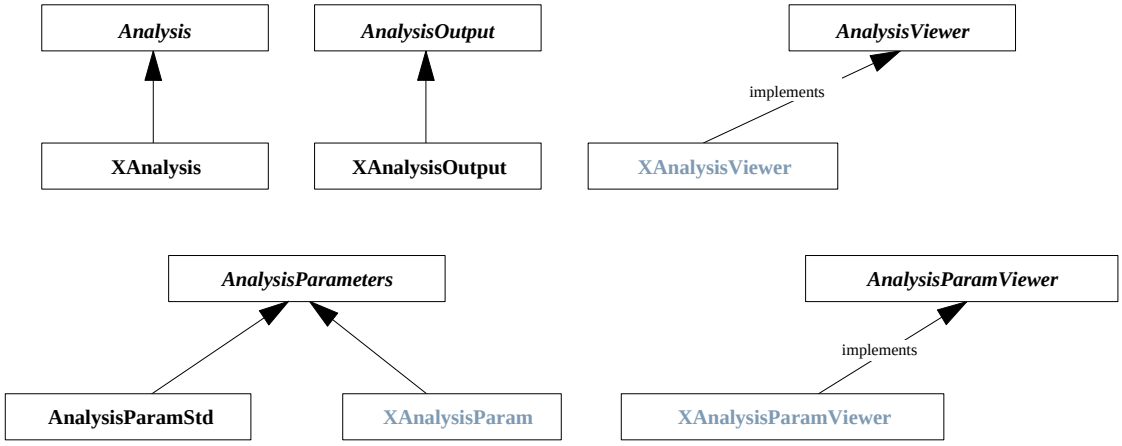
14.1.1 How to Create a Custom Analysis

To add your own analysis, create a subclass of Analysis, for example XAnalysis (see illustration below), that implements the *run()* method, and creates an AnalysisOutput class, XAnalysisOutput, that will format the results of your analysis class.

To display your results, Calypso provides a standard viewer, DefaultAnalysisViewer, which can display any report that is a spreadsheet-style table of values. If you wish to create a different type of display, then you can create a class that implements the AnalysisViewer interface. Whatever viewer you choose, you must register it in the Analysis Viewer Config window

Standard analysis parameters are specified in AnalysisParamStd. You may have to subclass AnalysisParamStd to add custom parameters, and create a custom parameters viewer that implements AnalysisParamViewer for editing these parameters.

The following class diagram shows you the classes involved in creating a custom analysis.



Class hierarchy diagram: Adding your own Analysis. Optional classes are shown in grey.

Overview of Steps

- Step 1 — Create analysis parameters if applicable
- Step 2 — Create an analysis parameters viewer if applicable
- Step 3 — Create an AnalysisOutput
- Step 4 — Create an Analysis
- Step 5 — Register the new Analysis

Step 1 — Creating Analysis Parameters

This step is only necessary if your analysis requires custom parameters.

Do the following for creating the analysis parameters:

1. Create a class named `tk.risk.<analysis_name>Param` which extends `com.calypso.tk.risk.AnalysisParamStd`.

This class will be invoked from your Analysis class.

2. To make the analysis parameters persistent, create a class named `tk.risk.sql.<analysis_name>Param` which extends `com.calypso.tk.risk.sql.AnalysisParamStdSQL`.

3. Register individual parameters using Main Entry > Configuration > System > Analysis Parameter Name.

Sample Code

- AnalysisParamStd samples in `calypsox/tk/risk/DEMOParam.java` and `calypsox/tk/risk/ABCParam.java`. The DEMO analysis requires a frequency to generate time buckets and a number of threads for multi-threading.
- AnalysisParamStdSQL samples in `calypsox/tk/risk/sql/DEMOParamSQL.java` and `calypsox/tk/risk/sql/ABCParamSQL.java`.

Step 2 — Creating an Analysis Parameters Viewer

This step is only necessary if your analysis requires custom parameters to be edited by the user.

Create a class named `apps.risk.<analysis_name>ParamViewer` which implements the interface `com.calypso.apps.risk.AnalysisParamViewer`.

This class will be invoked from your Analysis Parameters class.

Sample Code

- Samples in `calypsox/apps/risk/ABCParamViewer.java` and `calypsox/apps/risk/DEMOParamViewer.java`

Step 3 — Creating an AnalysisOutput

Create a class named `tk.risk.<analysis_name>Output` that extends `com.calypso.tk.risk.AnalysisOutput`.

Implement the following members and methods:

- A member to store the report. You can model the report as a Vector of TradeItem objects (each TradeItem object represents a row in the report). Note that you can subclass TradeItem as applicable.
- A method for building the report, for example *addItem()* to add rows to the report.
- Methods that will allow the AnalysisViewer to display, print, save, export, aggregate the report. For example, when using the DefaultAnalysisViewer you will implement the following methods:
  - *getNumberOfRows()* to return the number of rows in the report
  - *getNumberOfColumns()* to return the number of columns in the report
  - *getHeaderAt(int col)* to return the heading text for a given column
  - *getColumnClassAt(int col)* to return the datatype of the data for a given column
  - *getValueAt(int row, int col)* to return the value for a given cell
  - *getAggregationSource()* to return the aggregation criteria

This class will be invoked from your Analysis class.

Sample Code

- AnalysisOutput samples in `calypsox/tk/risk/ABCOutput.java` and `calypsox/tk/risk/DEMOOutput.java`.
- TradeItem samples in `calypsox/tk/risk/TradeABCItem.java` and `calypsox/tk/risk/DemoItem.java`

Step 4 — Creating an Analysis

The Analysis class actually performs the analysis.

Create a class named `tk.risk.<analysis_name>Analysis` that extends `com.calypso.tk.risk.Analysis`.

Implement the following methods:

- *run()* — Returns an AnalysisOutput object (your custom AnalysisOutput) that contains the results of the analysis. It takes the following parameters:
  - a Vector of trades to be analyzed
  - a valuation date
  - a PricingEnv containing market data
  - an AnalysisParameters object, AnalysisParamStd or your custom AnalysisParamStd
- *getParamNames()* — Returns a list of all the parameters required by the Analysis class.

This class will be invoked from `com.calypso.tk.risk.AnalysisUtil`.

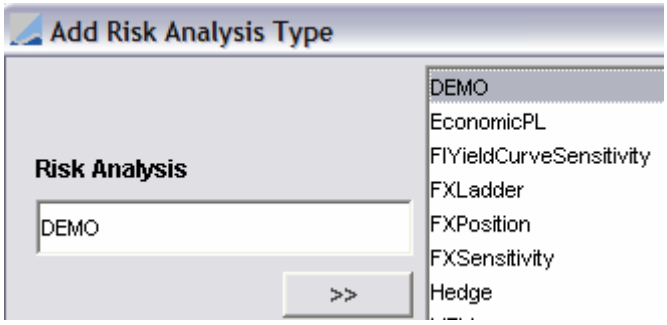
Sample Code

- Samples in `calypsox/tk/risk/ABCAnalysis.java` and `calypsox/tk/risk/DEMOAnalysis.java`. DEMO reports NPV and Break even rate for trades are grouped in user-defined time buckets by maturity date. The analysis makes use of the multi-threading capability of the system.

Step 5 — Registering the new Analysis

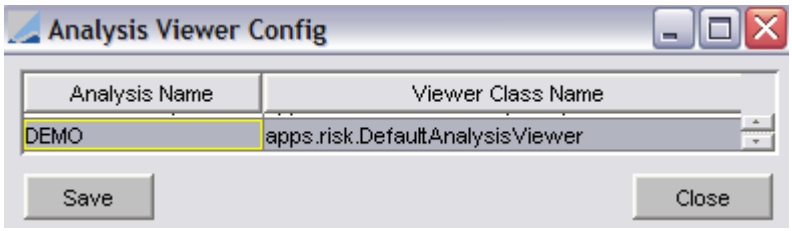
Do the following for registering the new analysis:

1. Add the analysis name to the riskAnalysis domain using the Add Risk Analysis Type window as shown below. This window is accessed from the Risk Analysis window when you click the ... button next to the Analysis Type field.



- Save and click "Update Domains" in the Risk Analysis window.

2. Associate the Analysis with an AnalysisViewer using Main Entry > Configuration > System > Analysis Viewer Configuration as shown below.



You can use the DefaultAnalysisViewer or create a custom AnalysisViewer as described in the following section.

14.1.2 How to Create a Custom AnalysisViewer

The default AnalysisViewer is `com.calypso.apps.risk.DefaultAnalysisViewer`.

Create a class named `apps.risk.<viewer_class_name>` which implements the interface `com.calypso.tk.risk.AnalysisViewer`.

This class will be invoked from `com.calypso.tk.risk.AnalysisViewerConfig`. You can associate the AnalysisViewer with a given Analysis using Main Entry > Configuration > System > Analysis Viewer Configuration.

14.1.3 How to Create a Custom Aggregation for an AnalysisViewer

Create a class named `tk.util.CustomAggregation` that implements the interface `com.calypso.tk.util.AggregationInterface`.

This class will be invoked from `com.calypso.tk.util.Aggregation`.

Sample Code

- Sample in `calypsox/tk/util/SampleCustomAggregation,java`

14.1.4 How to Create a Custom AnalysisHandler

Create a class named `tk.risk.<analysis_name>Handler` or `tk.risk.DefaultAnalysisHandler` that implements `com.calypso.tk.risk.AnalysisHandler`.

This class will be invoked from `com.calypso.tk.risk.AnalysisUtil` for exporting AnalysisOutput data to the AnalysisViewer.

14.1.5 How to Create a Custom Analysis Input Verifier

Analysis performs calculations over an AnalysisInput that contains the trades, the PricingEnv, etc. You can create a custom verifier of the input data.

Create a class named `tk.risk.<analysis_name>Verifier` or `tk.risk.CustomAnalysisVerifier` that implements `com.calypso.tk.risk.AnalysisVerifier`.

This class will be invoked from `com.calypso.tk.risk.AnalysisInput`.

14.1.6 How to Create Custom Template Keywords

Create a class named `tk.risk.<analysis_name>RiskFormatter` that implements `com.calypso.tk.risk.RiskFormatter`.

Analysis names are available in the riskAnalysis domain. Implement a *parse<keyword_name>()* method for each custom keyword.

This class will be invoked from `com.calypso.tk.risk.AnalysisUtil`.

14.2 How to Customize EconomicPLAnalysis

EconomicPLAnalysis explains the variation of Profit & Loss between two dates according to different effects such as Interest Rate change, Trade Amendment, etc. The explanation is based upon a Pricer Measure, varying one component of pricing while leaving all others unchanged.

The explanation is calculated within a calculator which can be product and/or effect specific. The calculator is defined by the interface `tk.risk.pl.PLCalculator`. Most calculation effects are implemented within `tk.risk.pl.DefaultPLCalculator`. Some products have their own PLCalculator, <product_type>PLCalculator.

Each effect is a column EconomicPLColumn in EconomicPLAnalysis. Existing columns are defined in the eco_pl_column domain. Each column is linked to an id and a PricerMeasure using the Economic PL Column window as shown below.

Name	Id	PricerMeasure	Description
MTM_PREVIOUS	1	NPV	MTM on Start Date
MTM_CURRENT	2	NPV	MTM on End Date
PL	3	NPV	Current MTM minus Previous MTM

The Economic PL Column window is invoked from the Economic PL Param Viewer when you click the "Create New Items" button.

To create a custom PLCalculator, create a class named `tk.risk.pl.<product_type>PLCalculator` which extends DefaultPLCalculator or an existing PLCalculator for a given product.

Implement the *process()* method.

This class will be invoked from `com.calypso.tk.risk.pl.PLCalculatorUtil`.

Sample Code

» Customizing a column from DefaultPLCalculator

```
public class MY_PRODUCTPLCalculator extends DefaultPLCalculator { }

public void process(EconomicPLCol column, Trade todayTrade,
      Trade yesterdayTrade, Pricer pricer,
      PricerMeasure[] measures, boolean isPositionB)
   throws PricerException {
      switch(column.getType()) {
         case CALYPSO_COLUMN_ID:{
            need to fill the  PricerMeasure[] measures
            pricer.price(…..measures)
         }
         break;
         default:super.process(column,todayTrade,yesterdayTrade,pricer,measures,
            isPositionB);
         break;
      }
   }
```

» Customizing a column from a product PLCalculator

```
public class MY_PRODUCTPLCalculator extends SwapPLCalculator { }
public void process(EconomicPLCol column, Trade todayTrade,
   Trade yesterdayTrade, Pricer pricer,
   PricerMeasure[] measures, boolean isPositionB)
   throws PricerException {
      switch(column.getType()) {
         case CALYPSO_COLUMN_ID:{
            need to fill the  PricerMeasure[] measures
            pricer.price(...measures)
          }
      break;
      default:super.process(column,todayTrade,yesterdayTrade,pricer,
         measures,
         isPositionB);
```

```
        break;
      }
   }
```

» Creating a custom column

Note - Your column id must be greater than 100 in order to avoid any conflict with Calypso ids.

The new column must be registered in the eco_pl_column domain, and in the Economic PL Column window.

```
public class MY PRODUCTPLCalculator extends DefaultPLCalculator { }

public void process(EconomicPLCol column, Trade todayTrade,
      Trade yesterdayTrade, Pricer pricer,
      PricerMeasure[] measures, boolean isPositionB)
   throws PricerException {
      switch(column.getType()) {
         case MY COLUMN ID:{
            need to fill the  PricerMeasure[] measures
         }
         break;
         default:super.process(column,todayTrade,yesterdayTrade,pricer,measures,
            isPositionB);
         break;
      }
}
```

14.3 How to Customize ScenarioAnalysis

ScenarioAnalysis allows defining market data scenarios to be applied to a set of trades, and calculates risk measures for those scenarios. You can create custom scenario market data, custom scenario rules, custom report viewers, and custom report viewer converters.

14.3.1 Creating a Custom Scenario Rule

Create a class named `tk.risk.ScenarioRule<name>` that implements `com.calypso.tk.risk.ScenarioRule`.

This class will be invoked from `com.calypso.apps.risk.ScenarioRulePanel`.

You need to register the custom rule name with the customScenarioRule domain.

Sample Code

» Sample in `calypsox/tk/risk/ScenarioRuleCustomZeroInterest.java`

14.3.2 Creating a Custom Scenario Market Data

Create a class named `tk.risk.CustomScenarioMarketData` that implements `com.calypso.tk.risk.CustomScenarioMarketDataInterface`.

This class will be invoked from `com.calypso.tk.risk.ScenarioMarketData`.

Sample Code

» Sample in `calypsox/tk/risk/SampleCustomScenarioMarketData.java`

14.3.3 Creating a Custom Report Viewer

Create a class named `tk.risk.<viewer>` that implements `com.calypso.tk.risk.ScenarioReportViewInterface`.

This class will be invoked from `com.calypso.tk.risk.ScenarioReportView` and `com.calypso.apps.risk.ScenarioReportViewWindow`.

The custom viewers must be registered in the domain ScenarioViewerClassNames.

14.3.4 Creating a Custom Report Viewer Converter

Create a class named `tk.risk.<viewer converter>` that implements `com.calypso.tk.risk.ScenarioReportViewConverterInterface`.

This class will be invoked from `com.calypso.tk.risk.ScenarioReportView` to convert the standard columns names to user-defined column names.

Sample Code

- Sample in `calypsox/tk/risk/ScenarioReportViewConverterSample.java`

14.3.5 Creating a Custom Notifcation Before/After Pricing a Trade

The interface CustomScenarioAnalysisInterface has been added, it has the following methods: *beforeApplyingAllRules()* and *afterApplyingAllRules()*.

Create a class named `tk.risk.DefaultCustomScenarioAnalysisInterface` that implements `com.tk.risk.CustomScenarioAnalysisInterface`.

This class will be invoked from ScenarioAnalysis.

Sample Code

- Sample in `calypsox/tk/risk/DefaultCustomScenarioAnalysisInterface.java`

14.4 Distributed Processing

14.4.1 How to Apply Distributed Processing to a Client Application

A client application will instantiate a DispatcherUser that connects to the Dispatcher for sending jobs and receiving the results.

For applying distributed processing to a risk analysis, see How to apply Distributed Processing to a Risk Analysis.

Overview of Steps

- Step 1 — Create a DispatcherJob
- Step 2 — Create a DispatcherJobOutput
- Step 3 — Implement DispatcherUserListener and instantiate DispatcherUser

Step 1 — Creating a DispatcherJob

A DispatcherJob is an individual job sent to the Dispatcher for calculation.

Create a class named `apps.distproc.<name>Job` or `tk.distproc.<name>Job` that extends `com.calypso.tk.distproc.DispatcherJob`.

Implement the *process()* method. The *process()* method will return its results as a DispatcherJobOutput.

Sample Code

- Sample in `calypsox/tk/distproc/DispatcherJobPricing.java`

Step 2 — Creating a DispatcherJobOutput

DispatcherJobOutput contains the results of DispatcherJob.

Create a class named `apps.distproc.<name>JobOutput` or `tk.distproc.<name>JobOutput` that extends `com.calypso.tk.distproc.DispatcherJobOutput`.

Implement *get()* and *set()* methods in your DispatcherJobOutput class that allow the calculation routine in your DispatcherJob to set the calculated result values. Usually, these result values are PricerMeasures.

This class will be invoked from your DispatcherJob.

Sample Code

» Sample in `calypsox/tk/distproc/DispatcherJobOutputPricing.java`

Step 3 — Implementing DispatcherUserListener and Instantiating DispatcherUser

Your client application must implement `com.calypso.tk.distproc.DispatcherUserListener`.

Any client of the distributed processing system must be able to:

- receive a large task from the user and divide that task into smaller jobs
- send the jobs (using a DispatcherUser)
- receive and relay the results
- assess completion of the jobs
- handle errors

Receiving and Dividing the Task

For example, you can use a Trade Filter to collect the trades that you want to process and create a DispatcherJob for every ten trades in the Trade Filter.

Also, a table of ratio by product type allows splitting the jobs through the classes `tk.distproc.SwaptionProductRatio` and `tk.distproc.CancellableSwapRatio`.

Currently, for a European swaption, the ratio is set to 5, and for a Bermudan swaption, the ratio is set to 10. Those settings can be modified in the above-mentioned classes.

Suppose the number of trades per job is set to 10:

- Each swap is counted for 1 trade. For 50 swaps in a portfolio, it will take 5 jobs to complete the task.
- Each European swaption is counted for 5 trades. For 50 swaptions in a portfolio, it will take 25 jobs to complete the task.
- Each Bermudan swaption is counted for 10 trades. For 50 swaptions in a portfolio, it will take 50 jobs to complete the task.

Sending Jobs

Your client application will create a DispatcherUser to send jobs to the Dispatcher. Your application will need a DispatcherConfig in order to specify the location of the running Dispatcher process, so that it can create the DispatcherUser as a client of the Dispatcher. In the line below, c is our DispatcherConfig object:

```
DispatcherUser d= new DispatcherUser(c);
```

The DispatcherUser can exist for the life of the client application, sending many DispatcherJobs via its *send()* method:

```
DispatcherUser.send(DispatcherJob);
```

When sending jobs, you can set the COMPRESS_DISPATCHER_JOBS environment property to true to compress the jobs, or to false otherwise. The default value is true. Note that on large jobs over a fast network, it may be faster to send the jobs uncompressed.

Receiving Results

The method for receiving results is established in the DispatcherUserListener interface:

```
public void jobFinished(DispatcherJobOutput out);
```

The *jobFinished()* method will receive and relay the results of the calculation when a job is finished. Inside *jobFinished()*, you should implement the work necessary to relay the results using a DispatcherJobOutput.

Assessing Completion

Most client applications will use a counter to count results as they arrive. Once the number of results equals the number of jobs sent, the client application can exit or proceed to other work.

Handling Errors

The method for handling errors is established in the DispatcherUserListener interface:

```
public void onDisconnect();
```

The *onDisconnect()* method is called if your connection to the Dispatcher is dropped. In this method you should implement error reporting, informing the user that the calculation has failed due to the lost connection.

Sample Code

- Sample in `calypsox/tk/distproc/BatchPricing.java`

14.4.2 How to Create a Custom Ratio Dispatcher by Product

Create a class named `tk.distproc.<product_type>ProductRatio` that implements the interface `tk.distproc.ProductRatio`.

This class will be invoked from `calypso.tk.distproc.ProductRatioUtil`.

Sample Code

- Sample in `calypsox/tk/distproc/SwapProductRatio.java`

14.4.3 How to Apply Distributed Processing to a Risk Analysis

Create a class named `tk.distproc.<analysis_name>AnalysisDispatcher` which implements `com.calypso.tk.distproc.AnalysisDispatcher`.

In order to detect errors returned by individual jobs that will make any further calculation meaningless, and to terminate the entire dispatch job, call the *DistAnalysis.stopAll()* method from *AnalysisDispatcher.taskFinished()*.

This class will be invoked from `com.calypso.tk.distproc.AnalysisDispatcher`.

Sample Code

- Sample in `com/calypso/tk/distproc/MTMAnalysisDispatcher.java`, `com/calypso/tk/distproc/MTMJob.java` and `com/calypso/tk/distproc/MTMJobOutput.java`
  - MTMAnalysisDispatcher — implements the split and aggregate methods which determine how the job is divided into tasks and how the result is aggregated.
  - MTMJob — extends DispatcherJob, runs the MTM analysis, and returns the output.
  - MTMJobOutput — extends DispatcherJobOutput, and returns the results from the MTMJob.

14.4.4 How to Create a Custom Error Notification

The default error notification is an email when the dispatcher or the calculator are disconnected from the system. Email server host and port come from `calpyso_mail.properties`. This means that such a file should exist in a directory that is in the classpath on all hosts involved: client host, dispatcher host, and all calculator hosts. The DispatcherConfig provides the "from" and "to" email addresses.

To use this email-on-errors feature, the Calculators must know what DispatcherConfig to use. That means that either:

- The Calculators are started with "-config <DispatcherConfig_name>" at the command line, or
- The Calculators are started in GUI mode, and the DispatcherConfig is chosen from the drop-down list.

To create a custom error notification, create a class named `tk.distproc.CustomErrorNotifier` that implements `com.calypso.tk.distproc.ErrorNotifier`.

This class will be invoked from `com.calypso.apps.distproc.DispatcherConfigWindow`, `com.calypso.tk.distproc.Calculator`, and `com.calypso.tk.distproc.Dispatcher`.

Section 15. Reference Data

15.1 Legal Entities

15.1.1 How to Create Custom Attributes on Legal Entities

You can build a custom input window. Your custom window can contain the fields you need in order to record the additional attributes. Users can then launch the window by clicking the **custom** button in the legal entity window

Create a class named `apps.refdata.LegalEntityCustomInputWindow` which implements the interface `com.calypso.apps.refdata.LegalEntityCustomInput`.

You must define the following methods in your LegalEntityCustomInputWindow class:

- *input()* — applies the display for your window.
- *allowAttributeWindow()* — specifies if you still allow Calypso's Legal Entity Attribute window to be launched.

This class will be invoked from `com.calypso.apps.refdata.BOLegalEntityWindow`.

Sample Code

» Sample in `calypsox/apps/refdata/SampleLegalEntityCustomInputWindow.java`

15.1.2 How to Apply Custom Validation to a LegalEntity

Create a class named `apps.refdata.CustomLegalEntityValidator` which implements the interface `com.calypso.apps.refdata.LegalEntityValidator`.

This class will be invoked from `com.calypso.apps.refdata.BOLegalEntityWindow`.

Sample Code

» Samples in `calypsox/apps/refdata/SampleCustomLegalEntityValidator.java` and `calypsox/apps/refdata/CustomLegalEntityValidatorWithLEAttributeCodeValidation.java`. CustomLegalEntityValidatorWithLEAttributeCodeValidation leaves a LegalEntity in the Disabled status as long as its attributes don't fit the requirements defined in the LegalEntityAttributeCode window.

15.1.3 How to Apply Custom Validation to a Legal Agreement

Create a class named `apps.refdata.CustomLegalAgreementValidator` that implements `com.calypso.apps.refdata.LegalAgreementValidator`.

This class will be invoked from `com.calypso.apps.refdata.BOLegalAgreementWindow`.

Sample Code

» Sample in `calypsox/apps/refdata/CustomLegalAgreementValidator.java`

15.1.4 How to Apply Custom Validation to a LegalEntity Contact

Create a class named `apps.refdata.CustomLEContactValidator` which implements the interface `com.calypso.apps.refdata.LEContactValidator`.

This class will be invoked from `com.calypso.apps.refdata.BOLEContactWindow`.

Sample Code

» Sample in calypsox/apps/refdata/SampleCustomLEContactValidator.java shows validation of the Swift code

15.1.5 How to Apply Custom Validation to LegalEntity Attributes

Create a class named apps.refdata.CustomLEAttributeValidator which implements the interface com.calypso.apps.refdata.LEAttributeValidator.

This class will be invoked from com.calypso.apps.refdata.BOLegalEntityAttributeWindow.

Sample Code

» Sample in calypsox/apps/refdata/SampleCustomLEAttributeValidator.java

15.1.6 How to Apply Custom Validation to LegalEntity Registrations

Create a class named apps.refdata.CustomRegistrValidator which implements the interface com.calypso.apps.refdata.RegistrValidator.

This class will be invoked from com.calypso.apps.refdata.BOLERegistrationWindow.

15.2 How to Apply Custom Validation to a Margin Call Config

Create a class named apps.refdata.CustomMarginCallConfigValidator which implements the interface com.calypso.apps.refdata.MarginCallConfigValidator.

This class will be invoked from com.calypso.apps.refdata.BOMarginCallConfigWindow.

Sample Code

» Sample in calypsox/apps/refdata/SampleCustomMarginCallConfigValidator.java

15.3 Settlement and Delivery Instructions (SDI)

15.3.1 How to Create a Custom SDI Selector

Refer to the *Calypso Settlements User Guide* for the methodology used by Calypso for selecting settlement and delivery instructions for trades.

Create a class named tk.bo.<product_type>SDISelector which implements the interface com.calypso.tk.bo.SDISelector.

This class will be invoked from com.calypso.tk.bo.SDISelectorUtil.

SDISelector contains the following methods:

- *validSDIList()* — returns a list of Valid SDI List.
- *validate()* — checks whether Static Data is still valid.
- *checkSettleDate()* — rejects any SDI record that uses an agent or intermediary that is not open for business on the transfer settlement date.
- *getSettlementMethod()* — returns the Settlement Method applicable for the Trade. The purpose is to restrict the search of Settlement Instruction based on a specific information set on the Trade. If the settlement Method returns null, the standard SDI search logic is applied.

- *getSettlementMethods()* — checks multiple transfer rules for ensuring that both pay and receive legs are using the same settlement method.
- *setTradeTransferRule()* — overrides, if necessary TradeTransferRule created by each BOProductHandler.
- *getValidManualSDIList()* — returns the first valid manual SDI or null if none found.
- *isBridgePossible()* — checks whether two Settle and Delivery Instructions are compatible. Compares the Settlement Methods of the Processing Organization and the Counterparty and returns whether Settlement Instructions are compatible.
- *matchSDIList()* — compares the two lists of Settle and Delivery Instructions and removes the incompatible SDI from the List. Both lists must be sorted by preference.
- *matchManualSDIList()* — same as above for Manual SDI.

Sample Code

» Sample in `calypsox/tk/bo/FXSDISelector.java`

15.3.2 How to Create a Custom SDI Sort Order

Create a class named `tk.refdata.CustomComparatorSDI` which implements the interface `java.util.Comparator`.

This class will be invoked from `com.calypso.tk.util.ComparatorFactory`.

Sample Code

» Sample in `calypsox/tk/refdata/CustomComparatorSDI.java`

15.3.3 How to Create a Custom SDI Description

Since users will be using the description to choose the proper instruction for making manual assignments of settlement and delivery instructions, you may wish to use custom descriptions.

Create a class named `tk.refdata.CustomSDIDescription` which implements the interface `com.calypso.tk.refdata.SDIDescription`.

This class will be invoked from `com.calypso.tk.refdata.SettleDeliveryInstruction` and `com.calypso.apps.refdata.BOSettlDeliveryWindow`.

Sample Code

» Sample in `calypsox/tk/refdata/CustomSDIDescription.java`

15.3.4 How to Apply a Custom Validation to an SDI

Create class named `apps.refdata.CustomSDIValidator` which implements the interface `com.calypso.apps.refdata.SDIValidator`.

This class will be invoked from `com.calypso.apps.refdata.BOSettlDeliveryWindow`.

Sample Code

» Sample in `calypsox/apps/refdata/SampleCustomSDIValidator.java`

15.3.5 How to add a Custom Menu Item to the SDI Window

Create a class named `apps.refdata.CustomSDIMenu` which implements the interface `com.calypso.apps.refdata.SDIMenu`.

This class will be invoked from com.calypso.apps.refdata.BOSettlDeliveryWindow for any utility function

Sample Code

- Sample in calypsox/apps/refdata/SDIMenu.java

15.3.6 How to Create a Custom Summary Panel on the SDI Window

Create a class named apps.refdata.CustomSDISummaryPanel which implements the interface com.calypso.apps.refdata.SDISummaryPanel.

This class will be invoked from com.calypso.apps.refdata.BOSettlDeliveryWindow.

Sample Code

- Sample in calypsox/apps/refdata/CustomSDISummaryPanel.java

15.3.7 How to Apply Custom Validation to an SDI Relationship

Create a class named apps.refdata.CustomSDIRelationShipValidator which implements the interface com.calypso.apps.refdata.SDIRelationShipValidator.

This class will be invoked from com.calypso.apps.refdata.SDIRelationShipWindow.

Sample Code

- Sample in calypsox/apps/refdata/SampleCustomSDIRelationShipValidator.java

15.3.8 How to Apply Custom Validation to a Manual SDI

Create a class named apps.refdata.ManualSDIValidator which implements the interface com.calypso.apps.refdataCustomManual.SDIValidator.

This class will be invoked from com.calypso.apps.refdata.ManualSDIWindow.

Sample Code

- Sample in calypsox/apps/refdata/SampleCustomManualSDIValidator.java

15.4 How to Apply Custom Validation to a Book

Create a class named apps.refdata.CustomBookValidator which implements the interface com.calypso.apps.refdata.BookValidator.

This class will be invoked from com.calypso.apps.refdata.BookWindow.

Sample Code

- Sample in calypsox/apps/refdata/CustomBookValidator.java

15.5 Static Data Filter

15.5.1 How to Create a Custom StaticDataFilter Attribute

Do the following for create a custom StaticDataFilter attribute:

1. Create a class named tk.refdata.CustomStaticDataFilter which implements the interface com.calypso.tk.refdata.StaticDataFilterInterface.

This class will be invoked from com.calypso.tk.refdata.StaticDataFilterElement and from com.calypso.tk.refdata.StaticDataMaintenanceElement.

2. Create a custom panel for the new attribute as described in the following section.

Sample Code

- Sample in calypsox/tk/refdata/CustomStaticDataFilter.java

15.5.2 How to Create a Custom Attribute Panel

Create a class named apps.refdata.<attribute_name>CustomAttributePanel that implements the interface com.calypso.apps.refdata.CustomAttributePanel.

This class will be invoked from com.calypso.apps.refdata.StaticDataFilterWindow.

Sample Code

- Sample in calypsox/apps/refdata/IS_NULLCustomAttributePanel.java

15.6 Trade Filter

15.6.1 Position Based Products

Added domain PositionBasedProducts - List of products that return true in their implementation of *isPositionBased()*. This list is used internally for excluding the trades whose products are position based from trade filters with that criteria.

This list is not to be modified, and should include at most all products that are position based products. Including products which return false from their *isPositionBased()* implementations will result in incorrect behavior when loaded through trade filters with the property setIncludePositionBased to false.

Not including all position based products in this list will only result in lower performance and higher memory requirements when loading trade filters with the property setIncludePositionBased to false.

15.6.2 How to Create a Custom Trade Filter Attribute

Do the following for creating a custom Trade Filter attribute:

1. Create a class named tk.mo.CustomCriterion<name> that implements com.calypso.tk.mo.CustomCriterion.

When implementing a custom attribute, you can decide whether to generate the SQL clause or not. The SQL clause will be appended to the Trade Filter SQL statement for loading trades. Generating the SQL clause allows loading trades more efficiently. However, in cases where generating the SQL clause is not feasible, you can let the *accept()* method in TradeFilter doing the filtering on the loaded trades.

This class will be invoked from com.calypso.tk.mo.CustomCriterion.

2. Register the new attribute in the customCriterion domain.

3. Create a custom panel for the new attribute as described in the following section.

Sample Code

- Samples in calypsox/tk/mo/CustomCriterionIssuer.java and calypsox/tk/mo/CustomCriterionFeeType.java.

  Issuer is implemented without generating the SQL clause, and FeeType generates the SQL clause.

15.6.3 How to Create a Custom Attribute Panel

Create a class named `apps.refdata.<attribute_name>Panel` that implements the interface `com.calypso.apps.refdata.CustomCriterionPanelInterface`.

This class will be invoked from `com.calypso.apps.refdata.TradeFilterWindow`.

15.7 How to Apply Custom Validation to a Fee Grid

Create a class named `apps.refdata.CustomFeeGridValidator` that implements `com.calypso.apps.refdata.FeeGridValidator`.

This class will be invoked from `com.calypso.apps.refdata.FeeGridWindow`.

15.8 CFD

15.8.1 How to Apply Custom Validation to a CFDContractDefinition

Create a class named `apps.refdata.CustomCFDContractValidator` which implements the interface `com.calypso.apps.refdata.CFDContractValidator`.

This class will be invoked from `com.calypso.apps.refdata.CFDContractWindow`.

Sample Code

- Sample in `calypsox/apps/refdata/SampleCustomCFDContractValidator.java`

15.8.2 How to Apply Custom Validation to a CFDCountryGrid

Create a class named `apps.refdata.CustomCFDCountryGridValidator` which implements the interface `com.calypso.apps.refdata.CFDCountryGridValidator`.

This class will be invoked from `com.calypso.apps.refdata.CFDCountryGridWindow`.

Sample Code

- Sample in `calypsox/apps/refdata/SampleCustomCFDCountryGridValidator.java`

15.9 Audit and Authorization

15.9.1 How to make a Class Auditable and Authorizable

Auditable means that all changes to an object such as INSERT, AMEND and REMOVE are recorded with the following information:

- what has changed in the object
- when was the object changed
- who made the changes

Authorizable means that when an object is changed, another user has to authorize the changes.

The activation of Audit and/or Authorization is done through the Admin Window. Refer to the *Calypso System Guide* for details.

Overview of Steps

- Step 1 — Create a class that implements Auditable or Authorizable (which in turn extends Auditable)
- Step 2 — Make the class persistent
- Step 3 —Register the class for audit and authorization

Step 1 — Creating a Class that Implements Auditable or Authorizable

Create a class that implements `com.calypso.tk.core.Auditable` for audit only, or `com.calypso.tk.core.Authorizable` for both audit and authorization.

Audit

Implement the following methods on Auditable:

- *doAudit()*
- *undo()*
- *clone()*
- *getVersion()*
- *setVersion()*

You also need to implement the following methods:

- *getUser()*
- *setUser()*

The following variables need to be defined:

- `int version`
- `String user`

Authorization

Implement the following methods on Authorizable:

- *getId()*
- *setId()*
- *diff()*
- *apply()*
- *getAuthName()*

The following variable needs to be defined:

- `int id`

Sample Code

» Sample in `samples.cookbook.tk.refdata.LegalEntityLimit.java`

Step 2 — Making the Class Persistent

To make the class persistent, create a class that extends `com.calypso.tk.core.sql.AuditSQL` for audit only, or `com.calypso.tk.core.sql.AuthorizableSQL` for both audit and authorization.

The following methods need to be overloaded in your SQL class:

- *save()*
- *remove()*
- *find()*

The database table does not need any field to store audit information. This is all taken care of by Calypso using `bo_audit`. However the table needs the following:

- a unique identifier, preferably an integer
- a version number (Calypso will control the versioning for you)

Sample Code

- Sample in `samples.cookbook.tk.refdata.sql.LegalEntityLimitSQL.java`
- Database scripts in `samples/cookbook/le_limit.sql` for Sybase and `samples/cookbook/le_limit_Oracle.sql` for Oracle

Step 3 — Registering the Class for Audit and Authorization

The class name should be registered in the classAuditMode domain for audit, and in the classAuthMode domain for authorization.

Sample Code

The LegalEntityLimit sample can be activated with the following additional files:

- `samples/cookbook/apps/refdata/LegalEntityLimitWindow.java`
- `samples/cookbook/tk/service/CustomDataServer.java`
- `samples/cookbook/tk/service/RemoteCustomData.java`
- `samples/cookbook/le_limit.sql` for Sybase or `samples/cookbook/le_limit_Oracle.sql` for Oracle

In order to run the sample, you will need to run the following at the command line to start the custom server:

```
rmic samples.cookbook.tk.service.CustomDataServer
```

When the Audit and Authorization modes are on, you will be able to save a LegalEntityLimit in the LegalEntityLimitWindow, and you will see that the values can be authorized using Main Entry > Processing > Data Authorization, and audited using Main Entry > Reports > Audit > Audit Report.

15.9.2 How to Create a Custom Authorization Window

The default Authorization Window is `com.calypso.apps.refdata.AuthorizationWindow`.

Create a class named `apps.refdata.<name>AuthViewer` that implements `com.calypso.apps.refdata.AuthViewer`.

This class will be invoked from `com.calypso.apps.refdata.AuthViewerUtil.`

15.9.3 How to add Custom Authorization to a Class

Create a class named `apps.util.<class_name>CheckAuthorization` that implements `com.calypso.apps.util.CheckAuthorization`.

This class will be invoked from `com.calypso.tk.refdata.AccessUtil`.

15.9.4 How to Create Custom Authorization Behavior

You can implement a custom authorization behavior that will be invoked when clicking the Accept button in the Authorization window. Create a class named `tk.refdata.CustomPreAuthorize` that implements `CustomPreAuthorizeInterface`.

15.10 Authentication

15.10.1 How to Create a Custom Authentication Mechanism

Create a class named `tk.service.CustomUserConnect` that implements the interface `com.calypso.tk.service.UserConnect`.

This allows using LDAP for example, as the authentication mechanism.

15.10.2 How to Create a Custom Password Validation Routine

Create a class named `tk.refdata.CustomPasswordValidator` which implements the interface `com.calypso.tk.refdata.CustomPasswordValidator`.

Sample Code

- Sample in `calypsox/tk/refdata/SampleCustomPasswordValidator.java`

15.11 Access Permissions

15.11.1 How to add Custom Access Permission Functions

To add a new function, add the function name to the function domain.

For example, we add MyCheck to the function domain.

Then in your custom code, you can check the function using:

```
If AccessUtil.isAuthorized('MyCheck')
```

15.11.2 How add Custom Access Permissions to a Class

Create a class named `apps.util.<class_name>CheckAccess` which implements the interface `com.calypso.tk.refdata.CheckAccess`.

Sample Code

- Samples in `calypsox/apps/util/SampleLEContactCheckAccess.java`, `calypsox/apps/util/BondAssetBackedCheckAccess.java` and `calypsox/apps/util/SampleSettleDeliveryInstructionCheckAccess.java`

15.11.3 How to Create Custom Trade Access Permissions

Create a class named `tk.refdata.CustomTradeAccess` that implements the interface `com.calypso.tk.refdata.TradeAccess`.

15.11.4 How to Create a Custom User Setup

To control which GUI and config properties for a given "reference user" should be duplicated when adding a new user or changing a user's group in the User Access Permission window.

Create a class named `apps.refdata.UserSetup` which implements the interface `com.calypso.apps.refdata.CustomUserSetup`.

Sample Code

» Sample in `calypsox/apps/refdata/UserSetup.java`

15.11.5 How to Apply Custom Validation to User Access Permissions

Create a class named `tk.refdata.DefaultCustomProfileValidator` which implements the interface `com.calypso.tk.refdata.CustomProfileValidator`.

Sample Code

» Sample in `calypsox/tk/refdata/DefaultCustomProfileValidator.java`

15.12 Scheduled Tasks

Scheduled Tasks can be used to run tasks on a regular basis, such as exporting data and and importing data on a regular basis.

Out-of-the-box, Calypso provides a number of Scheduled Tasks described in the *Calypso Trade Lifecycle User Guide*.

15.12.1 How to Create a Custom Scheduled Task

Create a class named `tk.util.ScheduledTask<name>` which extends the class `com.calypso.tk.util.ScheduledTask`.

Note - If you have a custom Scheduled Task that runs inside the Data Server, you need to use `trades = ScheduledTask.getTrades(ds, tf, null)` to load trades for a given trade filter. Do not use, `trades = ds.getRemoteTrade().getTrades(tf, null)`, or if you do, make sure to clone the Trade objects, otherwise the Trade cache will be corrupted.

Note - It is not recommended to audit every data member encapsulated within a custom ScheduledTask implementation. You should consider the following alternatives:

- If you are extending ScheduledTask, then override +doAudit:void. In this operation you can pick and choose the data members you wish audited.
- If you want simply exclude a data member from the audit process, rename the data member by pre-appending a double underscore to its moniker (for example, change class variable foo to __foo).

This class will be invoked from `com.calypso.tk.util.ScheduledTask`.

Sample Code

» Samples in `calypsox/tk/util/ScheduledTaskDEAL_REPORT.java` `calypsox/tk/util/ScheduledTaskFXNDF_CHECK.java`, `calypsox/tk/util/ScheduledTaskTASK_STAT_REPORT.java`, and `calypsox/tk/util/ScheduledTaskWAIT_STOP_ENGINE.java`.

15.12.2 How to Customize ScheduledTaskMESSAGE_MATCHING

The Scheduled Task MESSAGE_MATCHING can be used for matching external SWIFT messages. It can be customized in the following manner.

- com.calypso.tk.util.SwiftMessageInput — Write a class called CustomSwiftMessageInput which implements SwiftMessageInput.

If you do not write CustomSwiftMessageInput the scheduled task reads the text file Incomingswift.txt where the messages are separated by the separator specified in the scheduled task attributes.

- com.calypso.tk.util.swiftparser.TagParser — For parsing YYY tag you will have to write TagYYParser which implements TagParser.

- com.calypso.tk.util.swiftparser.MessageMatcher — For matching the "MT000" type of message you will need MessageMT000Matcher which implements MessageMatcher.

- com.calypso.tk.util.swiftparser.MessageProcessor — For processing the message MT000 (matched/unmatched) you will have to write MT000MessageProcessor which implements MessageProcessor. This class gets the BOMessage created using swift, and if matched then a BOMessage which is matched. Here you can do the final processing for that message.

Section 16. Workflow

16.1 Workflow Process

16.1.1 How to Create a Custom Exception Handler

Create a class named `tk.bo.workflow.exhandler.<exception_type>ExceptionHandler` which implements the interface `com.calypso.tk.bo.workflow.ExceptionHandler`.

This class will be invoked from `com.calypso.tk.bo.workflow.ExceptionHandlerUtil`.

Sample Code

- Sample in `calypsox/tk/bo/workflow/exhandler/EX_MISSING_SIExceptionHandler.java`

16.1.2 How to Create a Custom KickOffDate, CutOffDate

Create a class named `tk.bo.workflow.KickOffCalculator<config_name>` which implements the interface `com.calypso.tk.bo.workflow.KickOffCalculator`.

This class will be invoked from `com.calypso.tk.bo.workflow.KickOffCalculatorUtil` to override the KickOffDate and/or CutOffDate calculation for a particular KickOffCutOffConfig.

Note - For performance reasons, workflow rules are executed within the Data Server. Be very careful to clone any objects retrieved from the Data Server prior to modifying them.

Sample Code

- Sample in `calypsox/tk/bo/workflow/KickOffCalculatorTest.java`

16.1.3 How to Create Custom Data for a Task

Create a class named `tk.bo.CustomTaskInfo` which implements the interface `com.calypso.tk.bo.TaskFillInfo`.

This class will be invoked from `com.calypso.tk.bo.TaskFillInfoUtil`.

Sample Code

- Sample in `calypsox/tk/bo/SampleCustomTaskInfo.java`

16.1.4 How to Create Custom Rules, Actions, and Statuses

Do the following to create custom Rules, Actions, and Statuses for any workflow type:

1. Create a class named `tk.bo.workflow.rule.<component_name><workflow_type>Rule` that implements the interface `com.calypso.tk.bo.workflow.WfRule`.

This class will be invoked from `com.calypso.tk.bo.workflow.WorkflowRuleUtil`.

2. Register the new workflow type in the workflowType domain.

3. Add the new statuses, actions, and rules to this workflow type using Main Entry > Configuration > Workflow > Workflow Configuration > Domains > Entity as applicable.

See also How to add Custom Menu Items to the Task Station for adding custom actions.

16.2 How to Create a Custom Workflow Rule

To create a custom Trade Workflow Rule, create a class named `tk.bo.workflow.rule.<rule_name>TradeRule` which implements the interface `com.calypso.tk.bo.workflow.WfTradeRule`.

To create a custom Message Workflow Rule, create a class named `tk.bo.workflow.rule.<rule_name>MessageRule` which implements the interface `com.calypso.tk.bo.workflow.WfMessageRule`.

To create a custom Transfer Workflow Rule, create a class named `tk.bo.workflow.rule.<rule_name>TransferRule` which implements the interface `com.calypso.tk.bo.workflow.WfTransferRule`.

Note - The rule names need to be registered with the appropriate workflow rule domains: workflowRuleMessage, workflowRuleTrade, and workflowRuleTransfer.

The following methods need to be implemented on the interfaces WfTradeRule, WfTransferRule and WfMessageRule:

- *check()* — This method should only contain tests, and no object should be modified in this method. The reason is that if a given transition has more than one rule, the system will first call all the *check()* methods, and if all of them return true, it will call the *update()* methods, and then save any object as applicable.

  This method can be run on both client and data server sides. When applying a transition for saving/updating objects, the workflow will run on the server, but if a user wants to simulate a transition, the workflow runs on the client side.

  - When loading static data, you should use BOCache, LocalCache, and the remote services when possible. The workflow will know by itself on which side the code runs. The following code for example, can be run on both sides.

```
LegalEntity po = BOCache.getLegalEntity(dsCon, transfer.getProcessingOrg());
```

  - When loading active data, you should first check if you run on the client or server side - to know that you have to test if the dbCon is null or not.

    If it is not null, you run on the server side and you must use the SQL class. Otherwise, you have to use the remote services. Note that a DSConnection is never null, even on the server side. Therefore, the code should be like:

```
if (dbCon != null) {
   trade = TradeSQL.getTrade(id, (Connection)dbCon);
} else {
   trade = dsCon.getRemoteTrade().getTrade(id);
}
```

- *getDescription()* — This method will be called from the Workflow Config window to display information about the rule.

- *update()* — This method will be called by the system when all rules return true from the *check()* methods. You can modify object in this method. Note that this method will ALWAYS be run on the server side. Therefore, ONLY the dbCon can be used. For example, you can do:

```
TaskSQL.save(newTask, (Connection)dbCon);
```

  Moreover, if you want to save and publish new events inside a workflow rule, that must be done in the *update()* method. You have to create the event and add it to the events vector that is one of the arguments of the method. For example, you can do:

```
TaskSQL.save(newTask, (Connection)dbCon);
PSEventTask taskEvent = new PSEventTask();
taskEvent.setTask(newTask);
events.addElement(taskEvent);
```

  If you want to create exception tasks, it is recommended to create BOException objects and add them to the excps vector that is one of the arguments of the method. For example, you can do:

```
BOException boExcp = new BOException(tradeId,
   this.getClass().getName(),
   "Exception Message",
```

```
    BOException.INFORMATION);
    excps.addElement(boExcp);
```

These classes will be invoked from `com.calypso.tk.bo.workflow.WorkflowRuleUtil`.

Sample Code

- Multiple samples in `calypsox.tk.bo.workflow.rule`

16.3 How to Implement a Custom Workflow

Calypso offers the ability to implement a workflow for any entity. We will use the LegalEntity object to illustrate the implementation of a custom workflow.

An implementation using the Book class was also successfully implemented and tested. Note however that the current implementation stores the entity id as an integer. This raises some issues as to the feasibility to use classes which use a String identifier.

16.3.1 Entity

The object for which you want to implement a custom workflow must be identified as an Entity.

Implementing EntityObject

The object for which you want to implement a generic workflow must implement `com.calypso.tk.core.EntityObject`.

For example, the following code was added to the class `com.calypso.tk.core.LegalEntity` to become an EntityObject.

New imports are needed:

```
import java.sql.Connection;
import com.calypso.tk.core.sql.LegalEntitySQL;
```

The following methods provide a simple yet sufficient implementation of EntityObject interface:

```
/**
* New class field keeps a reference to EntityState
* @see com.calypso.tk.core.EntityState
*/
protected EntityState  _entityState = new EntityState();

/**
* Returns a unique id for this EntityObject. Note that together with
* the value returned by <code>getEntityType()</code>, the ID-type pair
* must uniquely identify this Entity Object in the system.<br>
* It is possible, however, for 2 or more EntityObjects to have the same
* id if they have different Entity Types.
* <p>
*/
public int  getEntityId()  { return getId(); }

/**
* Returns a type that uniquely identifies this EntityObject "type".
* Typically, the simplest way to implement this method is simply to
* return getClass().toString(). However, this is left as an implementation
* detail to permit more customization control.
*/
public String getEntityType()  { return "LegalEntity"; }

/**
* Returns an object that encapsulates the Workflow State for this
* <code>EntityObject</code>.
*
* @return the state associated to this entity object
```

```
* @see com.calypso.tk.core.EntityState
*/
public EntityState getEntityState() { return _entityState; }

/**
* Sets the object which encapsulates the Workflow State for this
* <code>EntityObject</code>
*
* @param state the workflow state to associate to this entity object.
* @see com.calypso.tk.core.EntityState
*/
public void setEntityState(EntityState state) { _entityState = state; }

/**
* Returns the Processing Org associated with this entity. Note
* that if not applicable, the method should return "ALL", preferably.
*
*/
public String getProcessingOrg()    { return "ALL"; }
```

It is also important to remember to update the **clone()**, **readExternal(...)**, and **writeExternal(...)** methods. It is straightforward but quite important to pass the information through RMI:

```
public void writeExternal(ObjectOutput out)
   throws IOException {
      ...
      boolean v=_entityState != null;
      out.writeBoolean(v);
      if(v) _entityState.writeExternal(out);
   }

public void readExternal(ObjectInput in)
   throws IOException,ClassNotFoundException {
      ...
      if(in.readBoolean()) {
         _entityState = new EntityState();
         try {_entityState.readExternal(in);}
         catch(Exception e) {
            Log.error(this, e);
               throw new IOException(e.getMessage());
            }
         }
   }
```

Lastly, you'll need to adjust the **serialVersionUID** field.

Implementing EntityPersistence

When an object goes through the workflow, the actual saving to the database is done via the EntityObject interface. The object goes through the workflow and, if no error is raised, the object and its state are saved to persistent data by calling *EntityObjectSQL.save(EntityObject, Connection)*. In order to properly persist your object at this point, you will need to implement the appropriate persistence class. For example, if you have the following class `calypsox.tk.mypackage.MyObject` which implements EntityObject, then you have to create `calypsox.tk.mypackage.sql.MyObjectSQL` that implements `com.calypso.tk.core.sql.EntityPersistence`. In doing so, you ensure that EntityObjectSQL is able to save, retrieve, remove your objects properly as the object goes through the workflow.

The changes to EntityObjectSQL are minimal. For this example, the changes are made to `com.calypso.tk.core.sql.LegalEntitySQL`. The idea is for the object's associated EntityState to be saved/removed/retrieved as needed. Hence, all those methods which retrieve the LegalEntity object from the database make a call as follows:

```
EntityObjectSQL.setEntityState(legalEntity, con);
```

Similarly, the following calls are used in the save and remove methods, respectively:

```
EntityObjectSQL.saveEntityState(legalEntity, con);
EntityObjectSQL.removeEntityState(legalEntity, con);
```

We have established an association between the LegalEntity, an entity object, and its associated EntityState, its state. In order to ensure data integrity we must be sure that the memory image for the object matches that in the database.

Modifying ReferenceDataServerImpl

You need to change the API for saving the object.

For this example, we change the *save(LegalEntity)* method. Currently, the save operation is invoked as follows:

```
int lid = LegalEntitySQL.save(legalEntity);
```

By changing this to the following, everything is handled in the workflow, including the actual "save" operation:

```
saveEntityObject(legalEntity);
int lid = legalEntity.getId();
```

Adding Workflow Rules

You might have to add workflow rules for triggering the workflow.

In this example, we have created a simple class `com.calypso.tk.bo.workflow.rule.CheckValidLegalEntityRule` that checks various properties of a Legal Entity to determine whether or not it is valid.

The workflow can have the following status: NONE, PENDING, or VERIFIED. The Action NEW creates the transition from NONE to PENDING. The Action AMEND, with STP flag on, links PENDING to VERIFIED with a call to this rule. Lastly, there is a link back from VERIFIED to PENDING also on AMEND so that any changes to the LegalEntity are validated back through the rule.

16.3.2 Domain Data

The following domain values should be added:

1. Add the entity to the workflowType domain. In our example, we add LegalEntity.

2. Create the domain workflowRule<entity>. In our example, it is workflowRuleLegalEntity.

3. Add the workflow rules that you have created to the workflowRule<entity> domain. In our example, we add CheckValid to the workflowRuleLegalEntity domain.

16.3.3 Workflow

Once the domain values have been set, the workflow can be configured using Main Entry > Configuration > Workflow > Workflow Configuration.

Make sure to add the actions, rules, and status codes as applicable using the menu items under Domain > Entity. You will be prompted to enter the entity (LegalEntity in our example).

We provide a sample configuration of the LegalEntity workflow. You need to start with a clean database and apply Demonstration Data in order to load the sample configuration.

16.4 Task Station

16.4.1 How to Create a Custom Action Task Handler

A custom action task handler allows adding custom processing (such as displaying a warning message, prompting the user to enter additional data, etc.), when a given action is applied.

Create a class named `tk.bo.workflow.TaskHandler<action>` that implements `com.calypso.tk.bo.workflow.TaskHandler`.

This class will be invoked from `com.calypso.tk.bo.workflow.TaskHandlerUtil`.

16.4.2 How to Create a Custom Summary in the Trade Panel

Create a class named apps.reporting CustomTradeSummaryPanel which implements the interface com.calypso.apps.reporting.TradeSummaryPanel.

This class will be invoked from com.calypso.apps.reporting.TaskStationJFrame.

Sample Code

- Sample in calypsox/apps/reporting/SampleCustomTradeSummaryPanel.java

16.4.3 How to Create a Custom Summary in the Message Panel

Create a class named apps.reporting.CustomMessageSummaryPanel which implements the interface com.calypso.apps.reporting.MessageSummaryPanel.

This class will be invoked from com.calypso.apps.reporting.TaskStationJFrame.

Sample Code

- Sample in calypsox/apps/reporting/SampleCustomMessageSummaryPanel.java

16.4.4 How to Create a Custom Summary in the Transfer Panel

Create class named apps.reporting.CustomTransferSummaryPanel which implements the interface com.calypso.apps.reporting.TransferSummaryPanel.

This class will be invoked from com.calypso.apps.reporting.TaskStationJFrame.

Sample Code

- Sample in calypsox/apps/reporting/SampleCustomTransferSummaryPanel.java

16.4.5 How to Create a Custom Summary in the Exception Panel

Create a class named apps.reporting.CustomExceptionSummaryPanel which implements the interface com.calypso.apps.reporting.ExceptionSummaryPanel.

This class will be invoked from com.calypso.apps.reporting.TaskStationJFrame.

Sample Code

- Sample in calypsox/apps/reporting/SampleCustomExceptionSummaryPanel.java

16.4.6 How to add Custom Menu Items to the Task Station

Create a class named apps.reporting.CustomTaskStationMenu which implements the interface com.calypso.apps.reporting.CustomTaskStationMenu.

To create a custom action to be applied on trades, transfers or messages, implement the methods *handleWorkflowAction()* and *isWorkflowActionImplemented()*.

This class will be invoked from com.calypso.apps.reporting.TaskStationUtil.

Sample Code

- Menu items samples in calypsox/apps/reporting/CustomTaskStationMenuMessage.java, calypsox/apps/reporting/CustomTaskStationMenuTrade.java, calypsox/apps/reporting/CustomTaskStationMenuTransfer.java, calypsox/apps/reporting/Tag72InputCustomTaskStationMenuMessage.java
- Action sample in calypsox/apps/reporting/CustomTaskStationMenuTrade.java and calypsox/apps/trading/AuthorizeTradeWindow.java

The Authorize trade action has been added to forbid the authorization of a manual amendment without checking the amended fields.

Insert the Authorize action in the needed transitions of your trade workflow. In this example, the Authorize action takes place between PENDING and VERIFIED. Each time an amendment is done manually, the trade goes to PENDING and another user needs to authorize it.

On the Task Station, select your trade and process it. Then choose the Authorize action. A popup window will prompt you to authorize or reject the trade.

16.4.7 How to Create Custom Columns in the Task Station

Create a class named `apps.reporting.CustomTaskStationColumn` which implements the interface `com.calypso.apps.reporting.CustomTaskStationColumn`.

This class will be invoked from `com.calypso.apps.reporting.TaskStationUtil`.

Sample Code

- Sample in `calypsox/apps/reporting/SampleCustomTaskStationColumn.java`

16.4.8 How to Apply Custom Validation to the Copy Message Panel

Create a class named `apps.reporting.CustomTSCopyMessageValidator` which implements the interface `com.calypso.apps.reporting.TSCopyMessageValidator`.

This class will be invoked from `com.calypso.apps.reporting.TSCopyMessagePanel`.

16.4.9 How to Create a Custom Copied Message

Create a class named `apps.reporting.CustomTSMessageHandler` that implements `com.calypso.apps.reporting.TSMessageHandler`.

This class will be invoked from `com.calypso.apps.reporting.TSCopyMessagePanel`.

16.4.10 How to Apply Custom Validation to the Assign Window

Create class named `apps.refdata.CustomTSTransferValidator` which implements the interface `com.calypso.apps.reporting.TSTransferValidator`.

This class will be invoked from `com.calypso.apps.reporting.TSAssignmentJFrame`.

16.4.11 How to Apply Custom Validation to the Netting Manager Window

Create a class named `apps.refdata.CustomTSNettingManagerValidator` that implements the interface `com.calypso.apps.reporting.TSTransferValidator`.

This class will be invoked from `com.calypso.apps.reporting.TSNettingManagerJFrame`.

16.4.12 How to Apply Custom Validation to the Split Panel

Create a class named `apps.reporting.CustomTSSplitTransferValidator` that implements `apps.reporting.TSSplitTransferValidator`.

This class will be invoked from `com.calypso.apps.reporting.TSSplitPanel`.

16.4.13 How to Create a Custom PO SDI Selection

You can override the PO SDI selection whenever the counterparty SDI is manually selected in the Netting Manager, Assign and Split windows. See the Tip section under How to Create a Custom BO Trade Display for details.

Section 17. Cache Framework

The Cache Framework allows any cache mechanism to be plugged-in at the data level. For example, accounts can be cached using the LRU cache mechanism, while postings can be cached using a custom cache mechanism.

Out-of-the-box, the following cache mechanisms are available:

- The perishable LRU cache — This is a fixed size LRU cache for perishable objects. Items in cache remain valid until they expire. If the cache is full after expired objects have been removed, the LRU algorithm described below is then applied. This is the default cache mechanism.

- The LRU (Least Recently Used) cache — This is a fixed size LRU cache. The cache has a maximum size specified when it is created. When an item is added to the cache, if the cache is already at the maximum size, the least recently used item is deleted, then the new item is added.

- The Hashtable cache — The cache has a maximum size specified when it is created. When an item is added to the cache, if the cache is already at the maximum size, the full cache is cleared, then the new item is added. You might want to use this mechanism when memory is not an issue, and you can set the limit at a high enough value so the cache will almost never reach its full capacity. In this case, using the HashtableCache might be worthwhile because there is a performance hit in using another cache mechanism.

- The LFU (Least Frequently Used) cache — The behavior is identical to that of HashtableCache until the cache gets full. If and when the cache gets full, all the items in cache are sorted based on their popularity, and the 10% least popular objects in the cache are removed.

  Popularity is defined as the number of requests for that particular object in cache, so a popular object will be requested more than an unpopular object.

Cache mechanisms are defined in the "cache" domain. You can assign cache mechanisms to data using the Cache panel of the Admin Window. Refer to the *Calypso System Guide* for details.

The following data are cache-aware:

- BOMessage
- BOIncomingMessage
- Account
- Manual Settlement and Delivery Instructions
- Legal Entity Contacts
- Legal Entities
- Market Data Curves
- Legal Agreements
- Volatility Surfaces
- Correlation Matrix
- BO Posting
- Product
- Task
- Transfer
- Trade

17.1 How to Add Caching to a Custom Object

Overview of Steps

- Step 1 — Identify what cache you wish to use for your custom object
- Step 2 — Implement Cacheable or Perishable on the custom object
- Step 3 — Update the cache in the SQL class of the custom object
- Step 4 — Register the cache with a CacheListener. This is an optional step if you want other caches to know about changes in your cache.

Step 1 — Identifying what Cache to use

Cache names are specified in `com.calypso.tk.core.CacheLimit` and you can display them using Admin > Cache as shown below.

Name	Current	Hit Ratio	Server Limit	Client Limit	Expiration	Implementation
Account	0	?	100000	10000	None	LFUCache
Book	0	?	100000	100000	None	LFUCache
CFD Country Grid	0	?	100000	100000	None	LFUCache
CRE	0	?	100000	10000	None	LFUCache
CashSettleInfo	0	?	100000	100000	None	LFUCache
CorrelationMatrix	0	?	100000	10000	None	LFUCache
CreditRating	0	?	100000	10000	None	LFUCache
Curve	0	?	100000	10000	None	LFUCache

For example, for a custom product, you will use the "Product" cache, `CacheLimit.CACHE_PRODUCT`.

Step 2 — Implementing Cacheable or Perishable

The custom object must implement `com.calypso.tk.util.cache.Cacheable`, or `com.calypso.tk.util.cache.Perishable` if you want to apply the PerishableLRU cache mechanism (Perishable extends Cacheable).

Implement the *getKey()* method which returns an object that must uniquely identify the object among others of the same class. In other words, if we assume that the cache contains 1,000,000 various objects of all kinds, it must be possible to locate any unique object by using the value pair made up of the implementing object's class and the object which is returned by the *getKey()* method.

A key object must uniquely identify an Object in the system within which this Cacheable object is used. The pair of the object class (*Object.getClass().toString()*) and a unique identifier make a suitable key.

For a Perishable object, implement the *isExpired()* method that returns true if the object has expired, or false otherwise.

Sample Code

```
public Object getKey() {
   if (__key == null)
      __key = new Integer(getId());
      return __key;
}

public boolean isExpired(long millis) {
   return (millis > BOCache.getTimeToLive(CacheLimit.CACHE_PRODUCT,
      DSConnection.getDefault()));
}
```

Step 3 — Updating the Cache

You first need to create the Cache object as shown in the example below:

```
protected static Cache _products= CacheUtil.makeCache(CacheLimit.CACHE_PRODUCT);
```

Then you have to update the cache under the following circumstances:

- When you save the object using *put(Cacheable)* on the Cache object, for example *_products.put(Product)*.
- When you remove the object using *remove(Object)* on the Cache object, for example *_products.remove(Product)*.
- When you retrieve the object using *get(Object)* on the Cache object. If the object is not in the cache, you will retrieve the object from the database and update the cache using *put(Cacheable)* on the Cache object.

Sample Code

```
static public Product getFromCache(int productId) {
   synchronized (_lock) {
      return (Product)_products.get(new Integer(productId));
   }
}

public static Product getProduct(String type, int productId, Connection con)
   throws PersistenceException {
      Product product= getFromCache(productId);
      if(product != null) {
         return product;
      }
      product = loadProduct(type,productId,con);
      if (product != null) {
         putInCache(product);
      }
   return product;
}
```

Step 4 — Registering the Cache with a CacheListener

This is an optional step if you want other caches to know about changes in your cache. For example, to clear a related cache when you cache is cleared.

Implement the *addCacheListener()* method on the Cache object in the SQL class of your custom object.

Sample Code

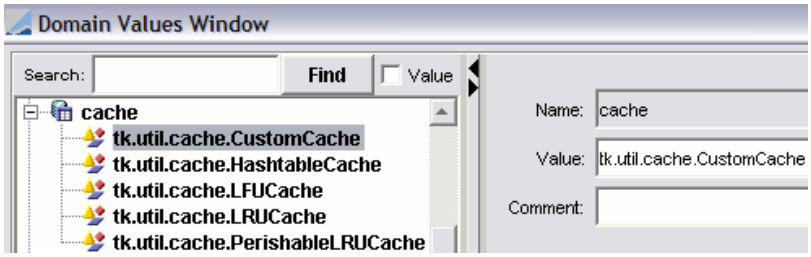
» Sample in `samples/TestCacheListener.java`. It provides a hook to listen to the Trade cache.

17.2 How to Create a Custom Cache Mechanism

Do the following for creating a cache mechanism:

1. Create a class named `tk.util.cache.<name>` that implements the interface `com.calypso.tk.util.cache.Cache`.

2. Register the cache mechanism in the cache domain.



The new cache mechanism will be available from the Cache panel in the Admin Window.

Sample Code

- Refer to `calypsox/tk/util/cache/CustomCache.java`

17.3 How to Disable Caching for a Given Object

Create a class named `tk.util.<cache_name>CacheValidator` that implements `com.calypso.tk.util.cache.CacheValidator`.

This class will be invoked from `com.calypso.tk.core.CacheUtil`.

Sample Code

- Sample in `calypsox/tk/util/cache/CurveCacheValidator.java`

Section 18. Administration

18.1 How to Create a Custom Login Dialog

Create a class named `tk.util.ClientVersion` that implements `com.calypso.tk.util.ClientVersionInterface`.

This class will be invoked from `com.calypso.apps.util.CalypsoLoginDialog`.

18.2 How to Create a Custom About Window

Create a class named `apps.main.ServerInfoDialog` that implements `com.calypso.apps.main.ServerInfoDialogInterface`.

This class will be invoked from `com.calypso.apps.main.ServerInfoDialog` (the dialog that appears under Main Entry > Help > About).

Sample Code

- Sample in `calypsox/apps/main/ServerInfoDialog.java`

18.3 How to Create Custom Keyboard Accelerators

Create a class named `apps.util.DefaultCustomListener` which implements the interface `com.calypso.apps.util.CustomListener`.

This class will be invoked from `com.calypso.apps.util.AppUtil`.

Sample Code

- Sample in `calypsox/apps/util/DefaultCustomListener.java` shows the binding of an action to the F1 key

18.4 How to Allow Custom Date Patterns

Create a class named `apps.util.CalypsoDateDocument` that implements `javax.swing.text.PlainDocument`.

This class is invoked from `com/calypso/apps/util/AppUtil`.

Sample Code

- Sample in `calypsox/apps/util/CalypsoDateDocument.java` to support date formats such as 14-Mar-05 and 14-Mar-2005.

18.5 How to Extend the Admin Window

Create a class named `apps.util.CustomExtendAdmin` that implements `com.calypso.apps.util.ExtendAdmin`.

This class will be invoked from `com.calypso.apps.util.AdminFrame`.

Sample Code

- Sample in `calypsox/apps/util/CustomExtendAdmin.java` shows adding a new menu item and a custom tab to the Admin window.

Section 19. Developer's Notes

These notes are intended for internal development only.

19.1 How to add a non-transient Attribute to an Externalizable Class

19.1.1 Release

You are adding the attribute under `release`.

1. Look at AUDIT_VERSION in `release.src.com.calypso.tk.core.CalypsoVersion`. For example, it is 60900.

2. In the *readExternal()* method of the Externalizable class in release, call *CalypsoVersion.checkAuditVersion(__auditVersion, 60900)*. This means that this attribute was added in version 60900.

19.1.2 Patch

You are patching the attribute into one code line.

1. Look at AUDIT_VERSION in `patch.com.calypso.tk.core.CalypsoVersion`. For example, it is 60009.

2. In the *readExternal()* method of the Externalizable class **in both release and patch**, call *CalypsoVersion.checkAuditVersion(__auditVersion, 60900, 60009)*. This means that this attribute was added in version 60900 and patched in 60009

You are patching the attribute into two code lines.

1. Look at AUDIT_VERSION in `patch.com.calypso.tk.core.CalypsoVersion`. For example, it is 60009.

2. In the *readExternal()* method of the Externalizable class **in release and both patches**, call *CalypsoVersion.checkAuditVersion(__auditVersion,* 71500, 70009, 60024*)*. This means that this attribute was added in version 71500, and patched in 70009 and 60024.

19.2 How to use the Comparator Factory

Do the following for creating a comparator class:

1. Create a class named `com.calypso.tk.util.Comparator<name>` that implements `java.util.Comparator`.

2. Register the class with ComparatorFactory as shown in the example below for ComparatorAuthorizable.

```
public static Comparator getAuthComparator(){
        if (_authComp == null){
            _authComp = new ComparatorAuthorizable();
        }
        return _authComp;
    }
```

3. Access the comparator using the ComparatorFactory methods as shown below.

```
Comparator comparator = ComparatorFactory.getAuthComparator ();
```

Index

D

E

F

G

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X